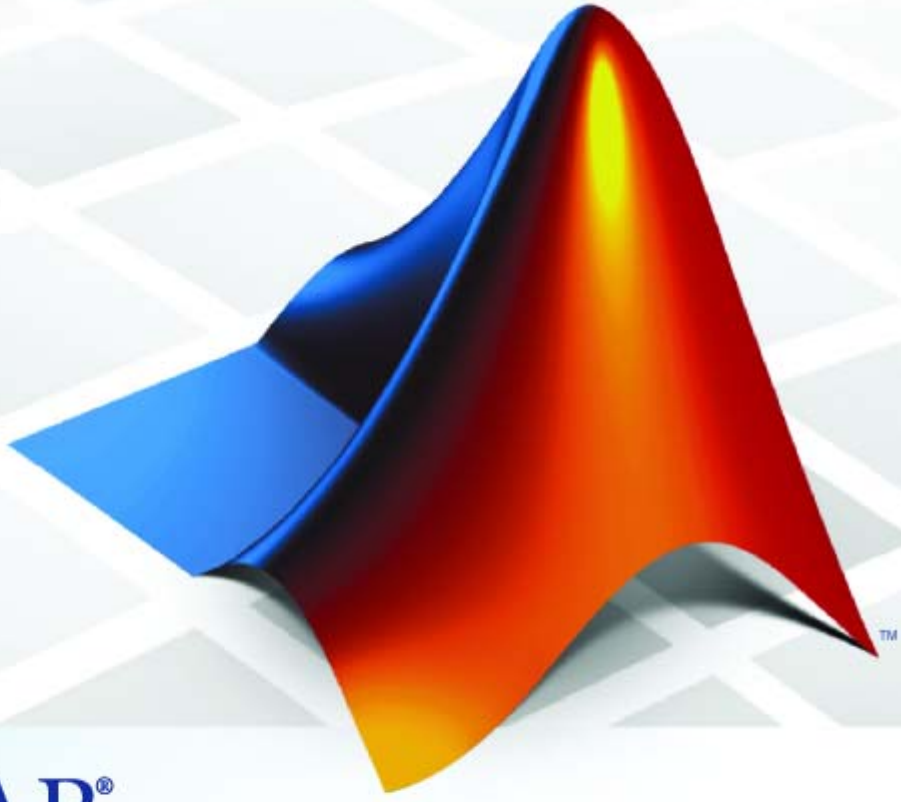
Real-Time Workshop® Embedded Coder™ 5
Reference



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Real-Time Workshop® Embedded Coder™ Reference

Trademarks

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

Patents

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

Revision History

September 2006	Online only	New for Version 4.5 (Release 2006b)
March 2007	Online only	Revised for Version 4.6 (Release 2007a)
September 2007	Online only	Revised for Version 5.0 (Release 2007b)
March 2008	Online only	Revised for Version 5.1 (Release 2008a)
October 2008	Online only	Revised for Version 5.2 (Release 2008b)
March 2009	Online only	Revised for Version 5.3 (Release 2009a)
September 2009	Online only	Revised for Version 5.4 (Release 2009b)
March 2010	Online only	Revised for Version 5.5 (Release 2010a)

Contents

1 Function Reference

2 Class Reference

3 Alphabetical List

4 Block Reference

Blocks — Alphabetical List

5

Configuration Parameters

6

Index

Function Reference

AUTOSAR

AUTOSAR Component Import (p. 1-2)	Control import of AUTOSAR components
AUTOSAR Configuration (p. 1-3)	Control and validate AUTOSAR configuration

AUTOSAR Component Import

arxml.importer	Construct `arxml.importer` object
createCalibrationComponentObjects (arxml.importer)	Create Simulink calibration objects from AUTOSAR calibration component
createComponentAsModel (arxml.importer)	Create AUTOSAR atomic software component as Simulink model
createComponentAsSubsystem (arxml.importer)	Create AUTOSAR atomic software component as Simulink atomic subsystem
createOperationAsConfigurableSubsystem (arxml.importer)	Create configurable Simulink subsystem library for client-server operation
getCalibrationComponentNames (arxml.importer)	Get calibration component names
getComponentNames (arxml.importer)	Get atomic software component names
getDependencies (arxml.importer)	Get list of XML dependency files
getFile (arxml.importer)	Return XML file name for `arxml.importer` object
setDependencies (arxml.importer)	Set XML file dependencies
setFile (arxml.importer)	Set XML file name for `arxml.importer` object

AUTOSAR Configuration

addIOConf (RTW.AutosarInterface)	Add AUTOSAR I/O configuration to model
attachToModel (RTW.AutosarInterface)	Attach `RTW.AutosarInterface` object to model
getComponentName (RTW.AutosarInterface)	Get XML component name
getDataTypePackageName (RTW.AutosarInterface)	Get XML data type package name
getDefaultConf (RTW.AutosarInterface)	Get default configuration
getExecutionPeriod (RTW.AutosarInterface)	Get runnable execution period
getImplementationName (RTW.AutosarInterface)	Get XML implementation name
getInitEventName (RTW.AutosarInterface)	Get initial event name
getInitRunnableName (RTW.AutosarInterface)	Get initial runnable name
getInterfacePackageName (RTW.AutosarInterface)	Get XML interface package name
getInternalBehaviorName (RTW.AutosarInterface)	Get XML internal behavior name
getIOAutosarPortName (RTW.AutosarInterface)	Get I/O AUTOSAR port name
getIODataAccessMode (RTW.AutosarInterface)	Get I/O data access mode
getIODataElement (RTW.AutosarInterface)	Get I/O data element name
getIOErrorStatusReceiver (RTW.AutosarInterface)	Get receiver port name

getIOInterfaceName (RTW.AutosarInterface)	Get I/O interface name
getIOPortNumber (RTW.AutosarInterface)	Get I/O AUTOSAR port number
getIOServiceInterface (RTW.AutosarInterface)	Get port I/O service interface
getIOServiceName (RTW.AutosarInterface)	Get port I/O service name
getIOServiceOperation (RTW.AutosarInterface)	Get port I/O service operation
getIsServerOperation (RTW.AutosarInterface)	Determine whether server is specified
getPeriodicEventName (RTW.AutosarInterface)	Get periodic event name
getPeriodicRunnableName (RTW.AutosarInterface)	Get periodic runnable name
getPortDefaultConf (RTW.AutosarInterface)	Get port default configuration
getServerInterfaceName (RTW.AutosarInterface)	Get name of server interface
getServerOperationPrototype (RTW.AutosarInterface)	Get server operation prototype
getServerPortName (RTW.AutosarInterface)	Get server port name
getServerType (RTW.AutosarInterface)	Determine server type
RTW.AutosarInterface	Construct `RTW.AutosarInterface` object
runValidation (RTW.AutosarInterface)	Validate `RTW.AutosarInterface` object against model
setComponentName (RTW.AutosarInterface)	Set XML component name

setInitEventName (RTW.AutosarInterface)	Set initial event name
setInitRunnableName (RTW.AutosarInterface)	Set initial runnable name
setIOAutosarPortName (RTW.AutosarInterface)	Set AUTOSAR port name
setIODataAccessMode (RTW.AutosarInterface)	Set I/O data access mode
setIODataElement (RTW.AutosarInterface)	Set I/O data element
setIOInterfaceName (RTW.AutosarInterface)	Set I/O interface name
setIsServerOperation (RTW.AutosarInterface)	Indicate that server is specified
setPeriodicEventName (RTW.AutosarInterface)	Set periodic event name
setPeriodicRunnableName (RTW.AutosarInterface)	Set periodic runnable name
setServerInterfaceName (RTW.AutosarInterface)	Set name of server interface
setServerOperationPrototype (RTW.AutosarInterface)	Specify operation prototype
setServerPortName (RTW.AutosarInterface)	Set server port name
setServerType (RTW.AutosarInterface)	Specify server type
syncWithModel (RTW.AutosarInterface)	Synchronize configuration with model

C++ Encapsulation Interface Control

attachToModel (RTW.ModelCPPClass)	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
getArgCategory (RTW.ModelCPPArgsClass)	Get argument category for Simulink model port from model-specific C++ encapsulation interface
getArgName (RTW.ModelCPPArgsClass)	Get argument name for Simulink model port from model-specific C++ encapsulation interface
getArgPosition (RTW.ModelCPPArgsClass)	Get argument position for Simulink model port from model-specific C++ encapsulation interface
getArgQualifier (RTW.ModelCPPArgsClass)	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface
getClassName (RTW.ModelCPPClass)	Get class name from model-specific C++ encapsulation interface
getDefaultConf (RTW.ModelCPPClass)	Get default configuration information for model-specific C++ encapsulation interface from Simulink model
getNumArgs (RTW.ModelCPPClass)	Get number of step method arguments from model-specific C++ encapsulation interface
getStepMethodName (RTW.ModelCPPClass)	Get step method name from model-specific C++ encapsulation interface
RTW.configSubsystemBuild	Configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem

RTW.getEncapsulationInterfaceSpecification	Get handle to model-specific C++ encapsulation interface control object
RTW.ModelCPPArgsClass	Create C++ encapsulation interface object for configuring model class with I/O arguments style step method
RTW.ModelCPPVoidClass	Create C++ encapsulation interface object for configuring model class with `void-void` style step method
runValidation (RTW.ModelCPPArgsClass)	Validate model-specific C++ encapsulation interface against Simulink model
runValidation (RTW.ModelCPPVoidClass)	Validate model-specific C++ encapsulation interface against Simulink model
setArgCategory (RTW.ModelCPPArgsClass)	Set argument category for Simulink model port in model-specific C++ encapsulation interface
setArgName (RTW.ModelCPPArgsClass)	Set argument name for Simulink model port in model-specific C++ encapsulation interface
setArgPosition (RTW.ModelCPPArgsClass)	Set argument position for Simulink model port in model-specific C++ encapsulation interface
setArgQualifier (RTW.ModelCPPArgsClass)	Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface
setClassName (RTW.ModelCPPClass)	Set class name in model-specific C++ encapsulation interface
setStepMethodName (RTW.ModelCPPClass)	Set step method name in model-specific C++ encapsulation interface

Code Generation Objectives Customization

addCheck (rtw.codegenObjectives.Objective)	Add checks
addParam (rtw.codegenObjectives.Objective)	Add parameters
excludeCheck (rtw.codegenObjectives.Objective)	Exclude checks
modifyInheritedParam (rtw.codegenObjectives.Objective)	Modify inherited parameter values
register (rtw.codegenObjectives.Objective)	Register objective
removeInheritedCheck (rtw.codegenObjectives.Objective)	Remove inherited checks
removeInheritedParam (rtw.codegenObjectives.Objective)	Remove inherited parameters
rtw.codegenObjectives.Objective	Create custom code generation objectives
setObjectiveName (rtw.codegenObjectives.Objective)	Specify objective name

Code Generation Verification

addCallback (cgv.CGV)	Add callback function
addConfigSet (cgv.CGV)	Add configuration set
addInputData (cgv.CGV)	Add input data
addPostLoadFiles (cgv.CGV)	Add files required by model
compare (cgv.CGV)	Compare signal data
configModel (cgv.Config)	Determine and change configuration parameter values
createToleranceFile (cgv.CGV)	Create file correlating tolerance information with signal names
displayReport (cgv.Config)	Display results of comparing configuration parameter values
getOutputData (cgv.CGV)	Get output data
getReportData (cgv.Config)	Return results of comparing configuration parameter values
getSavedSignals (cgv.CGV)	Display list of signal names to command line
plot (cgv.CGV)	Create plot for signal or multiple signals
run (cgv.CGV)	Execute CGV object
setOutputDir (cgv.CGV)	Specify folder
setOutputFile (cgv.CGV)	Specify output data file name

Function Prototype Control

addArgConf (RTW.ModelSpecificCPrototype)	Add argument configuration information for Simulink model port to model-specific C function prototype
attachToModel (RTW.ModelSpecificCPrototype)	Attach model-specific C function prototype to loaded ERT-based Simulink model
getArgCategory (RTW.ModelSpecificCPrototype)	Get argument category for Simulink model port from model-specific C function prototype
getArgName (RTW.ModelSpecificCPrototype)	Get argument name for Simulink model port from model-specific C function prototype
getArgPosition (RTW.ModelSpecificCPrototype)	Get argument position for Simulink model port from model-specific C function prototype
getArgQualifier (RTW.ModelSpecificCPrototype)	Get argument type qualifier for Simulink model port from model-specific C function prototype
getDefaultConf (RTW.ModelSpecificCPrototype)	Get default configuration information for model-specific C function prototype from Simulink model
getFunctionName (RTW.ModelSpecificCPrototype)	Get function name from model-specific C function prototype
getNumArgs (RTW.ModelSpecificCPrototype)	Get number of function arguments from model-specific C function prototype
getPreview (RTW.ModelSpecificCPrototype)	Get model-specific C function prototype code preview

RTW.configSubsystemBuild	Configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
RTW.getFunctionSpecification	Get handle to model-specific C prototype function control object
RTW.ModelSpecificCPrototype	Create model-specific C prototype object
runValidation (RTW.ModelSpecificCPrototype)	Validate model-specific C function prototype against Simulink model
setArgCategory (RTW.ModelSpecificCPrototype)	Set argument category for Simulink model port in model-specific C function prototype
setArgName (RTW.ModelSpecificCPrototype)	Set argument name for Simulink model port in model-specific C function prototype
setArgPosition (RTW.ModelSpecificCPrototype)	Set argument position for Simulink model port in model-specific C function prototype
setArgQualifier (RTW.ModelSpecificCPrototype)	Set argument type qualifier for Simulink model port in model-specific C function prototype
setFunctionName (RTW.ModelSpecificCPrototype)	Set function name in model-specific C function prototype

Model Entry Points

model_initialize	Initialization entry point in generated code for ERT-based Simulink model
model_SetEventsForThisBaseStep	Set event flags for multirate, multitasking operation before calling *model*_step for ERT-based Simulink model — not generated as of Version 5.1 (R2008a)
model_step	Step routine entry point in generated code for ERT-based Simulink model
model_terminate	Termination entry point in generated code for ERT-based Simulink model

Processor-in-the-Loop

Connectivity Configuration (p. 1-13)	Define processor-in-the-loop (PIL) configuration
Build (p. 1-13)	Configure PIL build process
Execution Download, Start and Stop (p. 1-14)	Control downloading, starting and resetting PIL executable on target hardware
Host and Target Communications (p. 1-14)	Configure host-target communications
Host-Side Communications (p. 1-14)	Configure host-side communications channel and drivers
Target-Side Communications (p. 1-14)	Configure target-side communications channel and drivers

Connectivity Configuration

rtw.connectivity.ComponentArgs	Provide parameters to each target connectivity component
rtw.connectivity.Config	Define connectivity implementation, comprising builder, launcher, and communicator components
rtw.connectivity.ConfigRegistry	Register connectivity configuration

Build

rtw.connectivity.MakefileBuilder	Configure makefile-based build process

Execution Download, Start and Stop

rtw.connectivity.Launcher	Control downloading, starting and resetting executable on target hardware

Host and Target Communications

rtIOStreamClose	Shut down communications channel with remote processor
rtIOStreamOpen	Initialize communications channel with remote processor
rtIOStreamRecv	Receive data from remote processor
rtIOStreamSend	Send data to remote processor

Host-Side Communications

rtiostream_wrapper	Test `rtiostream` shared library methods
rtw.connectivity.RtIOStreamHost-Communicator	Configure host-side communications

Target-Side Communications

rtw.pil.RtIOStreamApplication-Framework	Configure target-side communications

System Target File Callback Interface

slConfigUIGetVal	Return current value for custom target configuration option
slConfigUISetEnabled	Enable or disable custom target configuration option
slConfigUISetVal	Set value for custom target configuration option

Target Function Library Table Creation

addAdditionalHeaderFile	Add additional header file to array of additional header files for TFL table entry
addAdditionalIncludePath	Add additional include path to array of additional include paths for TFL table entry
addAdditionalLinkObj	Add additional link object to array of additional link objects for TFL table entry
addAdditionalLinkObjPath	Add additional link object path to array of additional link object paths for TFL table entry
addAdditionalSourceFile	Add additional source file to array of additional source files for TFL table entry
addAdditionalSourcePath	Add additional source path to array of additional source paths for TFL table entry
addConceptualArg	Add conceptual argument to array of conceptual arguments for TFL table entry
addEntry	Add table entry to collection of table entries registered in TFL table
copyConceptualArgsToImplementation	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
createAndAddConceptualArg	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry

createAndAddImplementationArg	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
createAndSetCImplementationReturn	Create implementation return argument from specified properties and add to implementation for TFL table entry
enableCPP	Enable C++ support for function entry in TFL table
getTflArgFromString	Create TFL argument based on specified name and built-in data type
registerCFunctionEntry	Create TFL function entry based on specified parameters and register in TFL table
registerCPPFunctionEntry	Create TFL C++ function entry based on specified parameters and register in TFL table
registerCPromotableMacroEntry	Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)
setNameSpace	Set name space for C++ function entry in TFL table
setReservedIdentifiers	Register specified reserved identifiers to be associated with TFL table
setTflCFunctionEntryParameters	Set specified parameters for function entry in TFL table
setTflCOperationEntryParameters	Set specified parameters for operator entry in TFL table

2

Class Reference

AUTOSAR

In this section...
"AUTOSAR Component Import" on page 2-1
"AUTOSAR Configuration" on page 2-1

AUTOSAR Component Import

arxml.importer	Control import of AUTOSAR components

AUTOSAR Configuration

RTW.AutosarInterface	Control and validate AUTOSAR configuration

C++ Encapsulation Interface Control

RTW.ModelCPPArgsClass	Control C++ encapsulation interfaces for models using I/O arguments style step method
RTW.ModelCPPClass	Control C++ encapsulation interfaces for models
RTW.ModelCPPVoidClass	Control C++ encapsulation interfaces for models using `void-void` style step method

Code Generation Objectives Customization

rtw.codegenObjectives.Objective	Customize code generation objectives

Code Generation Verification

cgv.CGV	Verify numerical equivalence of results
cgv.Config	Check and modify model configuration parameter values

Function Prototype Control

RTW.ModelSpecificCPrototype	Describe signatures of functions for model

3

Alphabetical List

addAdditionalHeaderFile

Purpose

Add additional header file to array of additional header files for TFL table entry

Syntax

```
addAdditionalHeaderFile(hEntry, headerFile)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

headerFile
: String specifying an additional header file.

Description

The `addAdditionalHeaderFile` function adds a specified additional header file to the array of additional header files for a TFL table entry.

Example

In the following example, the `addAdditionalHeaderFile` function is used along with `addAdditionalIncludePath`, `addAdditionalSourceFile`, and `addAdditionalSourcePath` to fully specify additional header and source files for a TFL table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

See Also

`addAdditionalIncludePath`, `addAdditionalSourceFile`, `addAdditionalSourcePath`

"Specifying Build Information for Function Replacements" in the Real-Time Workshop® Embedded Coder™ documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addAdditionalIncludePath

Purpose

Add additional include path to array of additional include paths for TFL table entry

Syntax

```
addAdditionalIncludePath(hEntry, path)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

path

String specifying the full path to an additional header file.

Description

The `addAdditionalIncludePath` function adds a specified additional include path to the array of additional include paths for a TFL table entry.

Example

In the following example, the `addAdditionalIncludePath` function is used along with `addAdditionalHeaderFile`, `addAdditionalSourceFile`, and `addAdditionalSourcePath` to fully specify additional header and source files for a TFL table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

See Also

`addAdditionalHeaderFile`, `addAdditionalSourceFile`, `addAdditionalSourcePath`

"Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addAdditionalLinkObj

Purpose

Add additional link object to array of additional link objects for TFL table entry

Syntax

```
addAdditionalLinkObj(hEntry, linkObj)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

linkObj
: String specifying an additional link object.

Description

The `addAdditionalLinkObj` function adds a specified additional link object to the array of additional link objects for a TFL table entry.

Example

In the following example, the `addAdditionalLinkObj` function is used along with `addAdditionalLinkObjPath` to fully specify an additional link object file for a TFL table entry.

```
% Path to external object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```

See Also

`addAdditionalLinkObjPath`

"Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Add additional link object path to array of additional link object paths for TFL table entry

Syntax

```
addAdditionalLinkObjPath(hEntry, path)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

path

String specifying the full path to an additional link object.

Description

The `addAdditionalLinkObjPath` function adds a specified additional link object path to the array of additional link object paths for a TFL table entry.

Example

In the following example, the `addAdditionalLinkObjPath` function is used along with `addAdditionalLinkObj` to fully specify an additional link object file for a TFL table entry.

```
% Path to external object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```

See Also

`addAdditionalLinkObj`

"Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addAdditionalSourceFile

Purpose

Add additional source file to array of additional source files for TFL table entry

Syntax

```
addAdditionalSourceFile(hEntry, sourceFile)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

sourceFile
: String specifying an additional source file.

Description

The `addAdditionalSourceFile` function adds a specified additional source file to the array of additional source files for a TFL table entry.

Example

In the following example, the `addAdditionalSourceFile` function is used along with `addAdditionalHeaderFile`, `addAdditionalIncludePath`, and `addAdditionalSourcePath` to fully specify additional header and source files for a TFL table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

See Also

`addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalSourcePath`

"Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addAdditionalSourcePath

Purpose

Add additional source path to array of additional source paths for TFL table entry

Syntax

```
addAdditionalSourcePath(hEntry, path)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

path
: String specifying the full path to an additional source file.

Description

The `addAdditionalSourcePath` function adds a specified additional source file path to the array of additional source file paths for a TFL table.

Example

In the following example, the `addAdditionalSourcePath` function is used along with `addAdditionalHeaderFile`, `addAdditionalIncludePath`, and `addAdditionalSourceFile` to fully specify additional header and source files for a TFL table entry.

```
% Path to external header and source files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');

op_entry = RTW.TflCOperationEntry;
.
.
.
addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));

addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

See Also

`addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalSourceFile`

"Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.addArgConf

Purpose

Add argument configuration information for Simulink model port to model-specific C function prototype

Syntax

```
addArgConf(obj, portName, category, argName, qualifier)
```

Description

`addArgConf(obj, portName, category, argName, qualifier)` method adds argument configuration information for a port in your ERT-based Simulink model to a model-specific C function prototype. You specify the name of the model port, the argument category (`'Value'` or `'Pointer'`), the argument name, and the argument type qualifier (for example, `'const'`).

The order of `addArgConf` calls determines the argument position for the port in the function prototype, unless you change the order by other means, such as the `RTW.ModelSpecificCPrototype.setArgPosition` method.

If a port has an existing argument configuration, subsequent calls to `addArgConf` with the same port name overwrite the previous argument configuration of the port.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype` or *obj* = `RTW.getFunctionSpecification(`*modelName*`)`.
portName	String specifying the unqualified name of an inport or outport in your Simulink model.
category	String specifying the argument category, either `'Value'` or `'Pointer'`.
argName	String specifying a valid C identifier.
qualifier	String specifying the argument type qualifier: `'none'`, `'const'`, `'const *'`, or `'const * const'`.

Examples

In the following example, you use the `addArgConf` method to add argument configuration information for ports `Input` and `Output` in an ERT-based version of `rtwdemo_counter`. After executing these commands, click the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to open the Model Interface dialog box and confirm that the `addArgConf` commands succeeded.

```
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a function control object
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the function control object to the model
attachToModel(a,gcs)
```

Alternatives

You can specify the argument configuration information in the Model Interface dialog box. See “Configuring Model Function Prototypes” in the Real-Time Workshop Embedded Coder documentation.

See Also

`RTW.ModelSpecificCPrototype.attachToModel`

“Controlling Generation of Function Prototypes” — Explains how to configure model function prototypes in generated code

cgv.CGV.addCallback

Purpose

Add callback function

Syntax

```
cgvObj.addCallback(CallbackFcn)
```

Description

`cgvObj.addCallback(CallbackFcn)` is an optional method that adds a callback function to the object. *`cgvObj`* is a handle to a `cgv.CGV` object. Before executing the model, for each input data, the object loads the input data (that you added using `addInputData`) and then calls the callback function.

The declaration of the callback function must receive the following parameters:

```
CallbackFcn(InputIndex, model_name, componentType,
connectivity)
```

`model_name`, *`componentType`*, *`connectivity`* are identical to the arguments that you specified in the *`cgvObj`* constructor. *`InputIndex`* is the index of the input data that the callback function executes.

You can specify only one callback for each object.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"
- "Using Callback Functions"

Purpose

Add checks

Syntax

`addCheck(obj, checkID)`

Description

`addCheck(obj, checkID)` includes the check, *`checkID`*, in the Code Generation Advisor. When a user selects the objective, the Code Generation Advisor includes the check, unless another objective with a higher priority excludes the check.

Input Arguments

`obj`	Handle to a code generation objective object previously created.
`checkID`	Unique identifier of the check that you add to the new objective.

Examples

Add the **Identify questionable code instrumentation (data I/O)** check to the objective.

```
addCheck(obj, 'Identify questionable code instrumentation (data I/O)');
```

See Also

`Simulink.ModelAdvisor`

How To

- "Creating Custom Objectives"
- "About IDs"

addConceptualArg

Purpose

Add conceptual argument to array of conceptual arguments for TFL table entry

Syntax

```
addConceptualArg(hEntry, arg)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

arg

Argument, such as returned by `arg = getTflArgFromString(name, datatype)`, to be added to the array of conceptual arguments for the TFL table entry.

Description

The `addConceptualArg` function adds a specified conceptual argument to the array of conceptual arguments for a TFL table entry.

Example

In the following example, the `addConceptualArg` function is used to add conceptual arguments for the output port and the two input ports for an addition operation.

```
hLib = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
```

```
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

See Also

`getTflArgFromString`

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

cgv.CGV.addConfigSet

Purpose

Add configuration set

Syntax

```
cgvObj.addConfigSet(configSet)
cgvObj.addConfigSet('configSetName')
cgvObj.addConfigSet('file','configSetFileName')
cgvObj.addConfigSet('file','configSetFileName','variable',
    'configSetName')
```

Description

`cgvObj.addConfigSet(configSet)` is an optional method that adds the configuration set to the object. *`cgvObj`* is a handle to a `cgv.CGV` object. *`cgvObj`*. *`configSet`* is a variable that specifies the configuration set.

`cgvObj.addConfigSet('configSetName')` is an optional method that adds the configuration set to the object. *`configSetName`* is a string that specifies the name of the configuration set in the workspace.

`cgvObj.addConfigSet('file','configSetFileName')` is an optional method that adds the configuration set to the object. *`configSetFileName`* is a string that specifies the name of the file that contains only one configuration set.

`cgvObj.addConfigSet('file','configSetFileName','variable', 'configSetName')` is an optional method that adds the configuration set to the object. The file contains one or more configuration sets. Specify the name of the configuration set to use.

This method replaces all configuration parameter values in the model with the values from the configuration set that you add. The object applies the configuration set when you call the `run` method. You can add only one configuration set for each `cgv.CGV` object.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"
- "Managing Configuration Sets"

Purpose

Add table entry to collection of table entries registered in TFL table

Syntax

```
addEntry(hTable, entry)
```

Arguments

hTable
: Handle to a TFL table previously returned by *hTable* = `RTW.TflTable`.

entry
: Handle to a function or operator entry that you have constructed after calling *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`

Description

The `addEntry` function adds a function or operator entry that you have constructed to the collection of table entries registered in a TFL table.

Example

In the following example, the `addEntry` function is used to add an operator entry to a TFL table after the entry is constructed.

```
hLib = RTW.TflTable;

% Create an entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u1','uint8');
```

```
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

See Also

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Add input data

Syntax

```
cgvObj.addInputData(InputIndex, 'DataFileName')
```

Description

`cgvObj.addInputData(InputIndex, 'DataFileName')` is a method that adds an input data file to the object. *`cgvObj`* is a handle to a `cgv.CGV` object. *`InputIndex`* is a unique numeric identifier to input data. The input data is used when the model executes. *`DataFileName`* is the name of the input file, with or without the `.mat` extension. If the input file is in the working folder, the *`cgvObj`* does not require the path.

The *`cgvObj`* uses the *`InputIndex`* to identify the input data associated with output data and output data files. *`cgvObj`* passes *`InputIndex`* to a callback function to identify the input data that the callback function uses.

Tips

- When calling `addInputData` you can modify configuration parameters by including their settings in the input file, *`DataFileName`*.
- `addInputData` does not qualify that the contents of *`DataFileName`* relate to the inputs of the model. Data that is not used by the model will not throw a warning or error.
- If you omit calling `addInputData` before executing the model, Code Generation Verification runs once using data in the base workspace.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

RTW.AutosarInterface.addIOConf

Purpose

Add AUTOSAR I/O configuration to model

Syntax

```
autosarInterfaceObj.addIOConf(SimulinkPort, DataAccessMode,
   autosarPort, InterfaceName, DataElement)
autosarInterfaceObj.addIOConf(SimulinkErrorStatusPort,
   'ErrorStatus', CorrespondingSimulinkReceiverPort)
autosarInterfaceObj.addIOConf(SimulinkBasicSoftwarePort,
   'BasicSoftwarePort', ServiceName, ServiceOperation,
   ServiceInterfacePath)
```

Description

You can designate inports and outports to be data sender/receiver ports, error status receivers, or access points to AUTOSAR Basic Software using the method `addIOConf`:

`autosarInterfaceObj.addIOConf(SimulinkPort, DataAccessMode, autosarPort, InterfaceName, DataElement)`

`autosarInterfaceObj.addIOConf(SimulinkErrorStatusPort, 'ErrorStatus', CorrespondingSimulinkReceiverPort)`

`autosarInterfaceObj.addIOConf(SimulinkBasicSoftwarePort, 'BasicSoftwarePort', ServiceName, ServiceOperation, ServiceInterfacePath)`

Each call adds an AUTOSAR I/O configuration to *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`SimulinkPort`	Inport/outport name (string)
`DataAccessMode`	Data access mode of the port. You can designate inports and outports to be data sender/receiver ports by specifying *`DataAccessMode`* to be one of the following:

	• `'ImplicitSend'` • `'ImplicitReceive'` • `'ExplicitSend'` • `'ExplicitReceive'` Use `'Implicit...'`where data is buffered by the run-time environment (RTE), or `'Explicit...'` where data is not buffered and hence not deterministic.
`autosarPort`	AUTOSAR port name (string)
`InterfaceName`	Interface name (string)
`DataElement`	Data element name (string)
`SimulinkErrorStatusPort`	The port you choose to receive error status.
`'ErrorStatus'`	The data access mode for ports chosen to be error status receivers.
`CorrespondingSimulinkReceiverPort`	The port that is listened to for error status. The data access mode for this port must be either `'ImplicitReceive'` or `'ExplicitReceive'`.

SimulinkBasicSoftwarePort	The port that you specify as an access point to AUTOSAR Basic Software.
'BasicSoftwarePort'	The data access mode for ports chosen to be access points to AUTOSAR Basic Software.
ServiceName	The service name you specify. Must be a valid AUTOSAR identifier.
ServiceOperation	The service operation you specify. Must be a valid AUTOSAR identifier.
ServiceInterfacePath	The service interface you specify. Must be a valid path of the form AUTOSAR/Service/*servicename*.

See Also

"Using the Configure AUTOSAR Interface Dialog Box" and "Configuring Ports for Basic Software and Error Status Receivers" in the Real-Time Workshop Embedded Coder documentation

Purpose

Add parameters

Syntax

`addParam(obj, paramName, value)`

Description

`addParam(obj, paramName, value)` adds a parameter to the objective, and defines the value of the parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**.

Input Arguments

obj	Handle to a code generation objective object previously created.
paramName	Parameter that you add to the objective.
value	Value of the parameter.

Examples

Add `Inlineparameters` to the objective, and specify the parameter value as `on`.

```
addParam(obj, 'InlineParams', 'on');
```

See Also

`get_param`

How To

- "Creating Custom Objectives"
- "Parameter Command-Line Information Summary"

cgv.CGV.addPostLoadFiles

Purpose

Add files required by model

Syntax

```
cgvObj.addPostLoadfiles({FileList})
```

Description

`cgvObj.addPostLoadfiles({FileList})` is an optional method that adds MATLAB® and MAT-files to the object. *`cgvObj`* is a handle to a `cgv.CGV` object. *`cgvObj`* executes and loads the files after opening the model and before running tests. *`FileList`* is a cell array of names of MATLAB and MAT-files in the testing directory that the model requires to run.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"
- "Using Callback Functions"

Purpose

Control import of AUTOSAR components

Description

You can use methods of the `arxml.importer` class to import AUTOSAR components in a controlled manner. For example, you can parse an AUTOSAR software component description file exported by DaVinci System Architect (from Vector Informatik Gmbh), and import the component into a Simulink model for subsequent configuration, code generation, and export to XML.

Construction

arxml.importer	Construct `arxml.importer` object

Methods

createCalibrationComponentObjects	Create Simulink calibration objects from AUTOSAR calibration component
createComponentAsModel	Create AUTOSAR atomic software component as Simulink model
createComponentAsSubsystem	Create AUTOSAR atomic software component as Simulink atomic subsystem
createOperationAsConfigurableSubsystem	Create configurable Simulink subsystem library for client-server operation
getCalibrationComponentNames	Get calibration component names
getComponentNames	Get atomic software component names
getDependencies	Get list of XML dependency files
getFile	Return XML file name for `arxml.importer` object

setDependencies	Set XML file dependencies
setFile	Set XML file name for `arxml.importer` object

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

Importing an AUTOSAR Software Component in the Real-Time Workshop Embedded Coder documentation

Purpose

Construct `arxml.importer` object

Syntax

```
importer_obj = importer(filename)
```

Description

`importer_obj = importer(filename)` constructs an `arxml.importer` object and parses the atomic software component described in the XML file specified by *`filename`*.

Note Only the atomic software components described in this XML file can be imported.

Input Arguments

`filename`	XML file containing description of atomic software component.

Output Arguments

`importer_obj`	Handle to newly created`arxml.importer` object.

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.attachToModel

Purpose

Attach `RTW.AutosarInterface` object to model

Syntax

```
autosarInterfaceObj.attachToModel(modelName)
```

Description

`autosarInterfaceObj.attachToModel(modelName)` attaches *`autosarInterfaceObj`*, an `RTW.AutosarInterface` object, to a loaded Simulink model with an ERT-based target.

Input Arguments

`modelName`	Name of a loaded Simulink model to which the object is going to be attached (string).

See Also

"Modifying and Validating an Existing AUTOSAR Interface" in the Real-Time Workshop Embedded Coder documentation

Purpose

Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model

Syntax

```
attachToModel(obj, modelName)
```

Description

`attachToModel(obj, modelName)` attaches a model-specific C++ encapsulation interface to a loaded ERT-based Simulink model.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.ModelCPPVoidClass`.
modelName	String specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ encapsulation interface modifications. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.attachToModel

Purpose

Attach model-specific C function prototype to loaded ERT-based Simulink model

Syntax

```
attachToModel(obj, modelName)
```

Description

`attachToModel(obj, modelName)` attaches a model-specific C function prototype to a loaded ERT-based Simulink model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype`.
modelName	String specifying the name of a loaded ERT-based Simulink model to which the object is going to be attached.

Alternatives

Click the **Configure Model Functions** button on the **Configuration Parameters > Real-Time Workshop > Interface** pane for flexible control over the model function prototypes that are generated for your model. Once you validate and apply your changes, you can generate code based on your function prototype modifications. See "Configuring Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Verify numerical equivalence of results

Description

Executes a model in different environments such as, simulation, Software-In-the-Loop (SIL), or Processor-In-the-Loop (PIL) and stores numerical results. Using the `cgv.CGV` class methods, you can create a script to verify that the model and the Real-Time Workshop Embedded Coder generated code produce numerically equivalent results.

`cgv.Config` and `cgv.CGV` use many of the same properties. When the `cgv.CGV` object executes, it uses `cgv.Config` to verify that the model is configured correctly for the mode of execution that you specify. If the top model is set to normal simulation mode, any reference models set to PIL mode will be changed to Accelerator mode.

Construction

`cgvObj` `= cgv.CGV('`*`model_name`*`')` creates a handle to a code generation verification object using the default properties. *`model_name`* is the name of the model that you are verifying. Returns a handle to a `cgv.CGV` object, *`cgvObj`*.

`cgvObj` `=`
`cgv.CGV('`*`model_name`*`','`*`PropertyName`*`','`*`PropertyValue`*`')`
constructs the object using options, specified as property name and value pairs. Property names and values are not case sensitive.

Properties

`ComponentType`

Define the SIL or PIL approach

If mode of execution is simulation (`connectivity` is `sim`), choosing either value for `ComponentType` has no effect on simulation results. However, `cgv.CGV` recommends configuration parameter values based on the value of `ComponentType`.

Value	Description
`'topmodel'` (default)	Top-model SIL or PIL simulation and standalone code interface mode.
`'modelblock'`	Model block SIL or PIL simulation and model reference Real-Time Workshop target code interface mode.

`connectivity`

Specify mode of execution

Value	Description
`'sim'` (default)	Mode of execution is Normal simulation. Recommends changes to a proper subset of the configuration parameters that all PIL targets require.
`'sil'`	Mode of execution is SIL. Recommends changes to the configuration parameters that SIL targets require.
`'tasking'`	Mode of execution is PIL using the Embedded IDE Link™ software and Altium® TASKING tools configuration. Requires that you specify `'processor'`. Recommends changes to the configuration parameters that PIL targets using the Embedded IDE Link software and Altium TASKING tools require.

Value	Description
`'custom'`	Mode of execution is PIL with custom connectivity that you provide using the PIL Connectivity API. Recommends changes to the configuration parameters that PIL targets with custom connectivity require.

`LogMode`

Specify how to save signal data

Value	Description
`'SignalLogging'` (default)	Save signal data to the MATLAB workspace variable during execution. To save the data, in the Configuration Parameters dialog box, the object uses the **Data Import/Export > Signal logging** parameter. In the **Signal logging** name field, define the variable name.
`'SaveOutput'`	Save signal data to the MATLAB workspace variable during execution. To save the data, in the Configuration Parameters Dialog box, the object uses the **Data Import/Export > Output** parameter. In the **Output**

Value	Description
	name field, define the variable name. When you use the **Output** parameter to save signal data, Simulink does not save bus outputs.

`processor`

Defines the processor type.

Use `processor` only when the value of `connectivity` is `tasking`.

Value	Description
`'ARM'`	ARM® processor
`'TriCore'`	Infineon® TriCore® processor
`'C166'`	Infineon® C166® processor
`'8051'`	Intel® 8051 processor
`'M16C'`	Renesas® M16C processor
`'DSP563xx'`	Freescale™ DSP566xx processor

`SaveModel`

Specify whether to save the model

Value	Description
`'off'` (default)	CGV checks the model configuration for compatibility with the connectivity target using `cgv.Config`. Does not save the model.
`'on'`	CGV updates and saves the model with required changes determined by `cgv.Config`. Saves the model in the working folder.

`ConfigModel`

Specify whether to disable configuration checks

Value	Description
`'on'` (default)	CGV checks the model configuration for compatibility with the connectivity target using `cgv.Config`.
`'off'`	Disable all use of `cgv.Config` on the model. Only specify `'off'` if you are sure the model is compatible.

`CheckInterface`

Specify whether to verify the model outports

Value	Description
`'on'` (default)	CGV checks the model outports for compatibility with CGV. CGV compiles the model first.
`'off'`	Disable verification of model outports. Only specify `'off'` if you are sure the model outports are compatible with CGV.

Methods

addCallback	Add callback function
addConfigSet	Add configuration set
addInputData	Add input data
addPostLoadFiles	Add files required by model
compare	Compare signal data
createToleranceFile	Create file correlating tolerance information with signal names
getOutputData	Get output data
getSavedSignals	Display list of signal names to command line
plot	Create plot for signal or multiple signals
run	Execute CGV object
setOutputDir	Specify folder
setOutputFile	Specify output data file name

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

The general workflow for testing a model for numerical equivalence using the `cgv.CGV` class is to:

1. Create a `cgv.CGV` object for each mode of execution and use the `cgv.CGV` set up methods to configure the model for each execution. The set up methods are:
   - `addCallBack`
   - `addConfigSet`
   - `addInputData`
   - `addPostLoadFiles`
   - `setOutputDir`
   - `setOutputFile`
2. Run the model for each mode of execution using `cgv.CGV.run`.
3. Use the `cgv.CGV` access methods to get and evaluate the data. The access methods are:
   - `getOutputData`
   - `getSavedSignals`
   - `plot`
   - `compare`

An object should only be run once. After the object is run, the set up methods are no longer used for that object. You then use the access methods for verifying the numerical equivalence of the results.

See Also

`cgv.Config`

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"
- Using Code Generation Verification

- “Verifying Code Using the SIL Simulation Mode”
- “Verifying Compiled Object Code with Processor-in-the-Loop Simulation”

Purpose

Check and modify model configuration parameter values

Description

Creates a handle to a `cgv.Config` object that supports checking and optionally modifying models for compatibility with various modes of execution, such as simulation, Software-In-the-Loop (SIL), or Processor-In-the-Loop (PIL).

To execute the model successfully in the mode that you specify, you might need to make additional modifications to the configuration parameter values or the model.

By default, `cgv.Config` modifies configuration parameter values to the value that it recommends, but does not save the model. Alternatively, you can specify that `cgv.Config` use one of the following approaches:

- Modify configuration parameter values to the values that `cgv.Config` recommends, and save the model. Specify this approach using the `SaveModel` property.
- List the values that `cgv.Config` recommends for the configuration parameters, but do not modify the configuration parameters or the model. Specify this approach using the `ReportOnly` property.

Do not use referenced configuration sets in models that you are modifying using `cgv.Config`. If the model uses a referenced configuration set, update the model with a copy of the configuration set. Use the `Simulink.ConfigSetRef.getRefConfigSet` method. For more information, see `Simulink.ConfigSetRef` in the Simulink documentation.

If you use `cgv.Config` on a model that executes a callback function, the callback function might modify configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. When this change occurs, the model might no longer be set up correctly for SIL or PIL. For more information, see "Using Callback Functions".

Construction

`cfgObj = cgv.Config('model_name')` creates a handle to a `cgv.Config` object, *`cfgObj`*, using default values for properties.

model_name is the name of the model that you are checking and optionally configuring.

cfgObj = cgv.Config('*model_name*','*PropertyName*','*PropertyValue*') constructs the object using options, specified as property name and value pairs. Property names and values are not case sensitive.

cgv.Config and cgv.CGV use many of the same properties. When the cgv.CGV object executes, it uses cgv.Config to verify that the model is configured correctly for the mode of execution that you specify.

Properties

ComponentType

Define the SIL or PIL approach

If mode of execution is simulation (connectivity is sim), choosing either value for ComponentType has no effect on simulation results. However, cgv.Config recommends configuration parameter values based on the value of ComponentType.

Value	Description
'topmodel' (default)	Top-model SIL or PIL simulation and standalone code interface mode.
'modelblock'	Model block SIL or PIL simulation and model reference Real-Time Workshop target code interface mode.

connectivity

Specify mode of execution

Value	Description
`'sim'` (default)	Mode of execution is simulation. Recommends changes to a proper subset of the configuration parameters that all PIL targets require.
`'sil'`	Mode of execution is SIL. Requires that the system target file is set to `'ert.tlc'` and that you do not use your own external target. Recommends changes to the configuration parameters that SIL targets require.
`'tasking'`	Mode of execution is PIL using the Embedded IDE Link software and Altium TASKING tools configuration. Requires that you specify `'processor'`. Recommends changes to the configuration parameters that PIL targets using the Embedded IDE Link software and Altium TASKING tools require.
`'custom'`	Mode of execution is PIL with custom connectivity that you provide using the PIL Connectivity API. Recommends changes to the configuration parameters that PIL targets with custom connectivity require.

`LogMode`

Specify how to save signal data

Value	Description
`'SignalLogging'` (default)	Save signal data to the MATLAB workspace variable during execution. To save the data, in the Configuration Parameters dialog box, the object uses the **Data Import/Export > Signal logging** parameter . In the **Signal logging** name field, define the variable name.
`'SaveOutput'`	Save signal data to the MATLAB workspace variable during execution. To save the data, in the Configuration Parameters dialog box, the object uses the **Data Import/Export > Output** parameter . In the **Output** name field, define the variable name. When you use the **Output** parameter to save signal data, Simulink does not save bus outputs.

`processor`

Defines the processor type

Use `processor` only when the value of `connectivity` is `tasking`.

Value	Description
'ARM'	ARM processor
'TriCore'	Infineon TriCore processor
'C166'	Infineon C166 processor
'8051'	Intel 8051 processor
'M16C'	Renesas M16C processor
'DSP563xx'	Freescale DSP566xx processor

`ReportOnly`

Specify whether to create a report

Specify whether to create a report of the values that `cgv.Config` recommends for the configuration parameters, without modifying the configuration parameters or the model. If you set `ReportOnly` to `'on'`, `SaveModel` must be `'off'`.

Value	Description
'off' (default)	Do not create a report.
'on'	Create a report.

`SaveModel`

Specify whether to save the model

Specify whether to modify configuration parameter values to the values that `cgv.Config` recommends, and save the model. If you set `SaveModel` to `'on'`, `ReportOnly` must be `'off'`.

Value	Description
'off' (default)	Do not save the model.
'on'	Save the model in the working folder.

Methods

configModel	Determine and change configuration parameter values
displayReport	Display results of comparing configuration parameter values
getReportData	Return results of comparing configuration parameter values

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Configure the `rtwdemo_iec61508` model for top-model SIL. Then view the changes at the MATLAB Command Window:

```
% Create a cgv.Config object and configure the model for top-model SIL.
cgvCfg = cgv.Config('rtwdemo_iec61508', 'LogMode', 'SaveOutput', ...
    'connectivity', 'sil');
cgvCfg.configModel();
% Display the results of what the cgv.Config object changed.
cgvCfg.displayReport();
% Close the rtwdemo_iec61508 model.
bdclose('rtwdemo_iec61508');
```

See Also

`cgv.CGV`

Tutorials

- "How To Verify a SIL or PIL Configuration"

How To

- "Configuring a PIL Simulation"
- "Choosing a PIL Approach"
- "Managing Configuration Sets"

Purpose

Compare signal data

Syntax

```
[matchNames, matchFigures, mismatchNames,
   mismatchFigures] =     cgv.CGV.compare(data_set1,
   data_set2)
[matchNames, matchFigures, mismatchNames,
   mismatchFigures] =     cgv.CGV.compare(data_set1,
   data_set2, 'Plot', 'param_value')
[matchNames, matchFigures, mismatchNames,
   mismatchFigures] =     cgv.CGV.compare(data_set1,
   data_set2, 'Plot', 'none', 'Signals', signal_list,
   'ToleranceFile', 'file_name.mat')
```

Description

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2)` compares data from two data sets which have common signal names between both executions. Possible outputs of the `cgv.CGV.compare` function are matched signal names, figure handles to the matched signal names, mismatched signal names, and figure handles to the mismatched signal names. By default, `cgv.CGV.compare` looks at all signals which have a common name between both executions.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', 'param_value')` compares all signals and plots the signals according to *param_value*.

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals', signal_list, 'ToleranceFile', 'file_name.mat')` compares only the given signals and produces no plots.

Input Arguments

`data_set1, data_set2`

Output data from a model. After running the model, use the `cgv.CGV.getOutputData` function to get the data. The cgv.CGV.getOutputData function returns a cell array of all output signal names.

`varargin`

Variable number of parameter name and value pairs.

varargin Parameters

You can specify the following argument properties for the `cgv.CGV.compare` function using parameter name and value argument pairs. These parameters are optional.

`Plot`(optional)

Designates which comparison data to plot. The value of this parameter must be one of the following:

- `'match'`: plot the comparison of the matched signals from the two datasets
- `'mismatch'`(default): plot the comparison of the mismatched signals from the two datasets
- `'none'`: do not produce a plot

`Signals`(optional)

A cell array of strings, where each string is a signal name in the `dataset`. Use `cgv.CGV.getSavedSignals` to view the list of available signal names in the *`dataset`*. *`signal_list`* can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
          'log_data.block_name.Data(:,2)',...
                      'log_data.block_name.Data(:,3)',...
                      'log_data.block_name.Data(:,4)'};
```

If a model component contains a space or newline character, MATLAB adds parantheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

If `Signals` is not present, all signals are compared.

`Tolerancefile`(optional)
: Name for the file created by the `cgv.CGV.createToleranceFile` function. The file contains the signal names and the associated tolerance parameter name and value pair for comparing the data.

Output Arguments

Depending on the data and the parameters, any of the following output arguments might be empty.

`match_names`

: Cell array of matching signal names.

`match_figures`

: Array of figure handles for matching signals

`mismatch_names`

: Cell array of mismatching signal names

`mismatch_figures`

: Array of figure handles for mismatching signals

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

cgv.Config.configModel

Purpose

Determine and change configuration parameter values

Syntax

```
cfgObj.configModel()
```

Description

`cfgObj.configModel()` determines the recommended values for the configuration parameters in the model. *`cfgObj`* is a handle to a `cgv.Config` object. The `ReportOnly` property of the object determines whether `configModel` reports or changes the configuration parameter values.

Tutorials

- "How To Verify a SIL or PIL Configuration"

How To

- "Configuring a PIL Simulation"
- "Managing Configuration Sets"

Purpose

Copy conceptual argument specifications to matching implementation arguments for TFL table entry

Syntax

```
copyConceptualArgsToImplementation(hEntry)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

Description

The `copyConceptualArgsToImplementation` function provides a quick way to copy conceptual argument specifications to matching implementation arguments. This function can be used when the conceptual arguments and the implementation arguments are the same for a TFL table entry.

Example

In the following example, the `copyConceptualArgsToImplementation` function is used to copy conceptual argument specifications to matching implementation arguments for an addition operation.

```
hLib = RTW.TflTable;

% Create an entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTflArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );
```

```
arg = hLib.getTflArgFromString('u1','uint8');
op_entry.addConceptualArg( arg );

arg = hLib.getTflArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

See Also

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry

Syntax

```
arg = createAndAddConceptualArg(hEntry, argType, varargin)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

argType
: String specifying the argument type to create: `'RTW.TflArgNumeric'` for numeric or `'RTW.TflArgMatrix'` for matrix.

varargin
: Parameter/value pairs for the conceptual argument. See varargin Parameters.

varargin Parameters

The following argument properties can be specified to the `createAndAddConceptualArg` function using parameter/value argument pairs. For example,

```
createAndAddConceptualArg(..., 'DataTypeMode', 'double', ...);
```

`Name`
: String specifying the argument name, for example, `'y1'` or `'u1'`.

`IOType`
: String specifying the I/O type of the argument: `'RTW_IO_INPUT'` for input or `'RTW_IO_OUTPUT'` for output. The default is `'RTW_IO_INPUT'`.

`IsSigned`
: Boolean value that, when set to `true`, indicates that the argument is signed. The default is `true`.

`WordLength`
: Integer specifying the word length, in bits, of the argument. The default is `16`.

`CheckSlope`
: Boolean flag that, when set to `true` for a fixed-point argument, causes TFL replacement request processing to check that the slope value of the argument exactly matches the call-site slope value. The default is `true`.

  Specify `true` if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify `false` if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

`CheckBias`
: Boolean flag that, when set to `true` for a fixed-point argument, causes TFL replacement request processing to check that the bias value of the argument exactly matches the call-site bias value. The default is `true`.

  Specify `true` if you are matching a specific [slope bias] scaling combination or a specific binary-point-only scaling combination on fixed-point operator inputs and output. Specify `false` if you are matching relative scaling or relative slope and bias values across fixed-point operator inputs and output.

`DataTypeMode`
: String specifying the data type mode of the argument: `'boolean'`, `'double'`, `'single'`, `'Fixed-point:  binary point scaling'`, or `'Fixed-point:  slope and bias scaling'`. The default is `'Fixed-point:  binary point scaling'`.

Note You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

`DataType`
: String specifying the data type of the argument: `'boolean'`, `'double'`, `'single'`, or `'Fixed'`. The default is `'Fixed'`.

`Scaling`
: String specifying the data type scaling of the argument: `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling. The default is `'BinaryPoint'`.

`Slope`
: Floating-point value specifying the slope of the argument, for example, `15.0`. The default is `1`.

  If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters

`SlopeAdjustmentFactor`
: Floating-point value specifying the slope adjustment factor (`F`) part of the slope, $F2^E$, of the argument. The default is `1.0`.

  If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter.

`FixedExponent`
: Integer value specifying the fixed exponent (`E`) part of the slope, $F2^E$, of the argument. The default is `-15`.

  If you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output, specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter.

`Bias`
: Floating-point value specifying the bias of the argument, for example, `2.0`. The default is `0.0`.

Specify this parameter if you are matching a specific [slope bias] scaling combination on fixed-point operator inputs and output.

`FractionLength`
: Integer value specifying the fraction length for the argument, for example, `3`. The default is `15`.

  Specify this parameter if you are matching a specific binary-point-only scaling combination on fixed-point operator inputs and output.

`BaseType`
: String specifying the base data type for which a matrix argument is valid, for example, `'double'`.

`DimRange`
: Dimensions for which a matrix argument is valid, for example, `[2 2]`. You can also specify a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means any two-dimensional matrix of size 2x2 or larger.

Returns

Handle to the created conceptual argument. Specifying the return argument in the `createAndAddConceptualArg` function call is optional.

Description

The `createAndAddConceptualArg` function creates a conceptual argument from specified properties and adds the argument to the conceptual arguments for a TFL table entry.

Examples

In the following example, the`createAndAddConceptualArg` function is used to specify conceptual output and input arguments for a TFL operator entry.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
```

```
                            'Name',       'y1', ...
                            'IOType',     'RTW_IO_OUTPUT', ...
                            'IsSigned',   true, ...
                            'WordLength', 32, ...
                            'FractionLength', 0);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
                            'Name',       'u1', ...
                            'IOType',     'RTW_IO_INPUT',...
                            'IsSigned',   true,...
                            'WordLength', 32, ...
                            'FractionLength', 0 );

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
                            'Name',       'u2', ...
                            'IOType',     'RTW_IO_INPUT',...
                            'IsSigned',   true,...
                            'WordLength', 32, ...
                            'FractionLength', 0 );
```

The following examples show some common type specifications using `createAndAddConceptualArg`.

```
% uint8:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
                            'Name',           'u1', ...
                            'IOType',         'RTW_IO_INPUT', ...
                            'IsSigned',       false, ...
                            'WordLength',     8, ...
                            'FractionLength', 0 );

% single:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
                            'Name',         'u1', ...
                            'IOType',       'RTW_IO_INPUT', ...
                            'DataTypeMode', 'single' );
```

```
% double:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'DataTypeMode', 'double' );

% boolean:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
                          'Name',         'u1', ...
                          'IOType',       'RTW_IO_INPUT', ...
                          'DataTypeMode', 'boolean' );

% Fixed-point using binary-point-only scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
                   'Name',           'y1', ...
                   'IOType',         'RTW_IO_OUTPUT', ...
                   'CheckSlope',     true, ...
                   'CheckBias',      true, ...
                   'DataTypeMode',   'Fixed-point: binary point scaling', ...
                   'IsSigned',       true, ...
                   'WordLength',     32, ...
                   'FractionLength', 28);

% Fixed-point using [slope bias] scaling:
createAndAddConceptualArg(hEntry, 'RTW.TflArgNumeric', ...
                   'Name',           'y1', ...
                   'IOType',         'RTW_IO_OUTPUT', ...
                   'CheckSlope',     true, ...
                   'CheckBias',      true, ...
                   'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                   'IsSigned',       true, ...
                   'WordLength',     16, ...
                   'Slope',          15, ...
                   'Bias',           2);
```

For examples of fixed-point arguments that use relative scaling or relative slope/bias values, see “Example: Creating Fixed-Point

Operator Entries for Relative Scaling (Multiplication and Division)" and "Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

createAndAddImplementationArg

Purpose

Create implementation argument from specified properties and add to implementation arguments for TFL table entry

Syntax

```
arg = createAndAddImplementationArg(hEntry, argType,
      varargin)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

argType
: String specifying the argument type to create: `'RTW.TflArgNumeric'` for numeric.

varargin
: Parameter/value pairs for the implementation argument. See varargin Parameters.

varargin Parameters

The following argument properties can be specified to the `createAndAddImplementationArg` function using parameter/value argument pairs. For example,

```
createAndAddImplementationArg(..., 'DataTypeMode', 'double', ...);
```

`Name`
: String specifying the argument name, for example, `'u1'`.

`IOType`
: String specifying the I/O type of the argument: `'RTW_IO_INPUT'` for input.

`IsSigned`
: Boolean value that, when set to `true`, indicates that the argument is signed. The default is `true`.

`WordLength`
: Integer specifying the word length, in bits, of the argument. The default is `16`.

`DataTypeMode`
: String specifying the data type mode of the argument: `'boolean'`, `'double'`, `'single'`, `'Fixed-point:  binary point scaling'`, or `'Fixed-point:  slope and bias scaling'`. The default is `'Fixed-point:  binary point scaling'`.

Note You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

`DataType`
: String specifying the data type of the argument: `'boolean'`, `'double'`, `'single'`, or `'Fixed'`. The default is `'Fixed'`.

`Scaling`
: String specifying the data type scaling of the argument: `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling. The default is `'BinaryPoint'`.

`Slope`
: Floating-point value specifying the slope of the argument, for example, `15.0`. The default is `1`.

  You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

`SlopeAdjustmentFactor`
: Floating-point value specifying the slope adjustment factor (`F`) part of the slope, $F2^E$, of the argument. The default is `1.0`.

  You can optionally specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter, but do not specify both.

`FixedExponent`
: Integer value specifying the fixed exponent (`E`) part of the slope, $F2^E$, of the argument. The default is `-15`.

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter, but do not specify both.

`Bias`
: Floating-point value specifying the bias of the argument, for example, `2.0`. The default is `0.0`.

`FractionLength`
: Integer value specifying the fraction length of the argument, for example, `3`. The default is `15`.

`Value`
: Constant value specifying the initial value of the argument. The default is 0.

  Use this parameter only to set the value of injected constant input arguments, such as arguments that pass fraction-length values or flag values, in an implementation function signature. Do not use it for standard generated input arguments such as `u1`, `u2`, and so on. You can place a constant input argument that uses this parameter at any position in the implementation function signature except as the return argument.

  You can inject constant input arguments into the implementation signature for any TFL table entry, but if the argument values or the number of arguments required depends on compile-time information, you should use custom matching. For more information, see "Refining TFL Matching and Replacement Using Custom TFL Table Entries" in the Real-Time Workshop Embedded Coder documentation.

Returns

Handle to the created implementation argument. Specifying the return argument in the `createAndAddImplementationArg` function call is optional.

Description

The `createAndAddImplementationArg` function creates an implementation argument from specified properties and adds the argument to the implementation arguments for a TFL table entry.

Note Implementation arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, or `boolean` (not fixed point data types).

Example

In the following example, the`createAndAddImplementationArg` function is used along with the `createAndSetCImplementationReturn` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                                  'Name',       'y1', ...
                                  'IOType',     'RTW_IO_OUTPUT', ...
                                  'IsSigned',   true, ...
                                  'WordLength', 32, ...
                                  'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',       'u1', ...
                              'IOType',     'RTW_IO_INPUT',...
                              'IsSigned',   true,...
                              'WordLength', 32, ...
                              'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',       'u2', ...
                              'IOType',     'RTW_IO_INPUT',...
                              'IsSigned',   true,...
```

```
                                'WordLength', 32, ...
                                'FractionLength', 0 );
```

The following examples show some common type specifications using `createAndAddImplementationArg`.

```
% uint8:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
                                'Name',           'u1', ...
                                'IOType',         'RTW_IO_INPUT', ...
                                'IsSigned',       false, ...
                                'WordLength',     8, ...
                                'FractionLength', 0 );

% single:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
                                'Name',         'u1', ...
                                'IOType',       'RTW_IO_INPUT', ...
                                'DataTypeMode', 'single' );

% double:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
                                'Name',         'u1', ...
                                'IOType',       'RTW_IO_INPUT', ...
                                'DataTypeMode', 'double' );

% boolean:
createAndAddImplementationArg(hEntry, 'RTW.TflArgNumeric', ...
                                'Name',         'u1', ...
                                'IOType',       'RTW_IO_INPUT', ...
                                'DataTypeMode', 'boolean' );
```

See Also

`createAndSetCImplementationReturn`

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

createAndSetCImplementationReturn

Purpose

Create implementation return argument from specified properties and add to implementation for TFL table entry

Syntax

```
arg = createAndSetCImplementationReturn(hEntry, argType,
      varargin)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by instantiating a TFL entry class, such as `hEntry = RTW.TflCFunctionEntry` or `hEntry = RTW.TflCOperationEntry`.

argType
: String specifying the argument type to create: `'RTW.TflArgNumeric'` for numeric.

varargin
: Parameter/value pairs for the implementation return argument. See varargin Parameters.

varargin Parameters

The following argument properties can be specified to the `createAndSetCImplementationReturn` function using parameter/value argument pairs. For example,

```
createAndSetCImplementationReturn(..., 'DataTypeMode', 'double', ...);
```

`Name`
: String specifying the argument name, for example, `'y1'`.

`IOType`
: String specifying the I/O type of the argument: `'RTW_IO_OUTPUT'` for output.

`IsSigned`
: Boolean value that, when set to `true`, indicates that the argument is signed. The default is `true`.

`WordLength`
: Integer specifying the word length, in bits, of the argument. The default is `16`.

`DataTypeMode`
: String specifying the data type mode of the argument: `'boolean'`, `'double'`, `'single'`, `'Fixed-point:  binary point scaling'`, or `'Fixed-point:  slope and bias scaling'`. The default is `'Fixed-point:  binary point scaling'`.

Note You can specify either `DataType` (with `Scaling`) or `DataTypeMode`, but do not specify both.

`DataType`
: String specifying the data type of the argument: `'boolean'`, `'double'`, `'single'`, or `'Fixed'`. The default is `'Fixed'`.

`Scaling`
: String specifying the data type scaling of the argument: `'BinaryPoint'` for binary-point scaling or `'SlopeBias'` for slope and bias scaling. The default is `'BinaryPoint'`.

`Slope`
: Floating-point value specifying the slope for a fixed-point argument, for example, `15.0`. The default is `1`.

  You can optionally specify either this parameter or a combination of the `SlopeAdjustmentFactor` and `FixedExponent` parameters, but do not specify both.

`SlopeAdjustmentFactor`
: Floating-point value specifying the slope adjustment factor (`F`) part of the slope, $F2^E$, of the argument. The default is `1.0`.

  You can optionally specify either the `Slope` parameter or a combination of this parameter and the `FixedExponent` parameter, but do not specify both.

`FixedExponent`
: Integer value specifying the fixed exponent (`E`) part of the slope, $F2^E$, of the argument. The default is `-15`.

You can optionally specify either the `Slope` parameter or a combination of this parameter and the `SlopeAdjustmentFactor` parameter, but do not specify both.

`Bias`
: Floating-point value specifying the bias of the argument, for example, `2.0`. The default is `0.0`.

`FractionLength`
: Integer value specifying the fraction length of the argument, for example, `3`. The default is `15`.

Returns

Handle to the created implementation return argument. Specifying the return argument in the `createAndSetCImplementationReturn` function call is optional.

Description

The `createAndSetCImplementationReturn` function creates an implementation return argument from specified properties and adds the argument to the implementation for a TFL table.

Note Implementation return arguments must describe fundamental numeric data types, such as `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, or `boolean` (not fixed point data types).

Example

In the following example, the `createAndSetCImplementationReturn` function is used along with the `createAndAddImplementationArg` function to specify the output and input arguments for an operator implementation.

```
op_entry = RTW.TflCOperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                                  'Name',       'y1', ...
                                  'IOType',     'RTW_IO_OUTPUT', ...
```

```
                                    'IsSigned',   true, ...
                                    'WordLength', 32, ...
                                    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                                'Name',       'u1', ...
                                'IOType',     'RTW_IO_INPUT',...
                                'IsSigned',   true,...
                                'WordLength', 32, ...
                                'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                                'Name',       'u2', ...
                                'IOType',     'RTW_IO_INPUT',...
                                'IsSigned',   true,...
                                'WordLength', 32, ...
                                'FractionLength', 0 );
```

The following examples show some common type specifications using `createAndSetCImplementationReturn`.

```
% uint8:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
                                'Name',           'y1', ...
                                'IOType',         'RTW_IO_OUTPUT', ...
                                'IsSigned',       false, ...
                                'WordLength',     8, ...
                                'FractionLength', 0 );

% single:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
                                'Name',         'y1', ...
                                'IOType',       'RTW_IO_OUTPUT', ...
                                'DataTypeMode', 'single' );

% double:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
```

```
                                    'Name',         'y1', ...
                                    'IOType',       'RTW_IO_OUTPUT', ...
                                    'DataTypeMode', 'double' );

% boolean:
createAndSetCImplementationReturn(hEntry, 'RTW.TflArgNumeric', ...
                                    'Name',         'y1', ...
                                    'IOType',       'RTW_IO_OUTPUT', ...
                                    'DataTypeMode', 'boolean' );
```

See Also

`createAndAddImplementationArg`

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create Simulink calibration objects from AUTOSAR calibration component

Syntax

```
importerObj.createCalibrationComponentObjects(componentName)
[success] = createCalibrationComponentObjects(importerObj.
com
   ponentName, CreateSimulinkObject, 'true')
```

Description

`importerObj.createCalibrationComponentObjects(componentName)` creates Simulink calibration objects from an AUTOSAR calibration component. This imports all your parameters into the Workspace and you can then assign them to block parameters in your Simulink model.

Input Arguments

componentName	Absolute short name path of calibration parameter component.
CreateSimulink Object, 'true'	Optional property/value pair. The property CreateSimulinkObject can be either true or false (default is true). If it is true, then: [*success*] = createCalibrationComponentObjects(*importerObj*.*componentName*, CreateSimulinkObject, 'true') creates the Simulink.AliasType and Simulink.NumericType corresponding to the AUTOSAR data types described in the XML file imported by *importerObj*.

Output Arguments

success	True if function is successful. False otherwise.

Examples

```
importer_obj.createCalibrationComponentObjects('/package/autosar_component2')
```

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

arxml.importer.createComponentAsModel

Purpose

Create AUTOSAR atomic software component as Simulink model

Syntax

```
[modelH,
   success] = importerObj.createComponentAsModel(ComponentName
   )
[modelH,
   success] = importerObj.createComponentAsModel(ComponentName
   , Property1, Value1, Property2, Value2, ...)
```

Description

`[modelH, success] = importerObj.createComponentAsModel(ComponentName)` creates a Simulink model corresponding to the AUTOSAR atomic software component `'COMPONENT'` described in the XML file imported by the `arxml.importer` object *`importerObj`*.

You can also specify optional property/value pairs when creating this Simulink model:

```
[modelH, success] =
importerObj.createComponentAsModel(ComponentName,
Property1, Value1, Property2, Value2, ...)
```

Input Arguments

`ComponentName`	Absolute short name path of the atomic software component.
`PropertyN, ValueN`	Optional property/value pairs. You can specify values for the following properties: `CreateSimulinkObject` `'true'` (default) or `'false'`. If `'true'`, then the function creates the `Simulink.AliasType` and `Simulink.NumericType` corresponding to the AUTOSAR data types in the XML file.

`NameConflictAction`
: `'overwrite'` (default) or `'makenameunique'` or `'error'`. Use this property to determine the action if a Simulink model with the same name as the component already exists.

`AutoSave`
: `'true'` or `'false'` (default). If `'true'`, then the function automatically saves the generated Simulink model.

Output Arguments

modelH	Model handle.
success	True if the function is successful. Otherwise, it is false.

Examples

```
importer_obj.createComponentAsModel('/package/autosar_component2')
```

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

arxml.importer.createComponentAsSubsystem

Purpose

Create AUTOSAR atomic software component as Simulink atomic subsystem

Syntax

```
[susbsysH,
   success] = importerObj.createComponentAsSubsystem(Component
   Name)
[susbsysH,
   success] = importerObj.createComponentAsSubsystem(Component
   Name, Property1, Value1, Property2, Value2, ...)
```

Description

`[susbsysH, success] = importerObj.createComponentAsSubsystem(ComponentName)` creates a Simulink subsystem corresponding to the AUTOSAR atomic software component `'COMPONENT'` described in the XML file imported by the `arxml.importer` object *`importerObj`*.

You can also specify optional property/value pairs when creating this Simulink subsystem:

`[susbsysH, success] = importerObj.createComponentAsSubsystem(ComponentName, Property1, Value1, Property2, Value2, ...)`

You can perform AUTOSAR configuration and code generation on atomic subsystems or function call subsystems. These subsystems must be convertible to model reference blocks by using the method:

```
Simulink.SubSystem.convertToModelReference
```

Note The AUTOSAR target automatically checks that the subsystem meets this requirement when you perform a subsystem build.

You do not have to convert your subsystem to a model reference block; it is optional. If you convert your subsystem to a referenced model, you can configure AUTOSAR options within the referenced model.

You can *export functions* for a single function-call subsystem. First configure your function-call subsystem AUTOSAR options (e.g., using the GUI from the Configuration Parameters dialog or by calling `autosar_gui_launch(subsystemName)`). Then right-click the subsystem and select **Real-Time Workshop > Export Functions**.

Input Arguments

`ComponentName`
: Absolute short name path of the atomic software component .

`PropertyN, ValueN`
: Optional property/value pairs. You can specify values for the following properties:

`CreateSimulinkObject`
: Boolean value, true or false (default is true). If true, the function creates the `Simulink.AliasType` and `Simulink.NumericType` corresponding to the AUTOSAR data types in the XML file.

`NameConflictAction`
: `'overwrite'` (default) or `'makenameunique'` or `'error'` . Use this property to determine the action to take if a Simulink model with the same name as the component already exists.

`AutoSave`
: Boolean value, true or false (default is false). If true, the function automatically saves the generated Simulink model.

arxml.importer.createComponentAsSubsystem

Output Arguments

susbsysH	Subsystem handle.
success	True if the function is successful. Otherwise, it is false.

Examples

```
importer_obj.createComponentAsSubsystem('/package/autosar_component2')
```

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create configurable Simulink subsystem library for client-server operation

Syntax

```
[modelH,
   success] = importerObj.createOperationAsConfigurableSubsyst
   ems(interfaceName)
[modelH,
   success] = importerObj.createOperationAsConfigurableSubsyst
   ems(InterfaceName, Property1, Value1, Property2, Value2,
   ...)
```

Description

`[modelH, success] = importerObj.createOperationAsConfigurableSubsystems(interfaceName)` creates a configurable Simulink subsystem library corresponding to the AUTOSAR client-server interface `'INTERFACE'`. This interface is described in the XML file imported by the `arxml.importer` object *`importerObj`*.

You can also specify optional property/value pairs when creating this Simulink subsystem library:

`[modelH, success] = importerObj.createOperationAsConfigurableSubsystems(InterfaceName, Property1, Value1, Property2, Value2, ...)`

Input Arguments

`interfaceName`	Absolute short name path of the client-server interface.
`PropertyN, ValueN`	Optional property/value pairs. You can specify values for the following properties: `CreateSimulinkObject` True (default) or false. If true, then the function creates the `Simulink.AliasType` and `Simulink.NumericType` corresponding

to the AUTOSAR data types in the XML file.

`NameConflictAction`
: `'overwrite'` (default) or `'makenameunique'` or `'error'`. Use this property to determine the action if a Simulink model with the same name as the component already exists.

`AutoSave`
: True or false (default). If true, then the function automatically saves the generated Simulink subsystem library.

`ForceClientBlkForBSP`
: True or false (default). If true, an Invoke AUTOSAR Server Operation block is created for a single argument operation that accesses Basic Software.

Output Arguments

modelH	Model handle.
success	True if the function is successful. False otherwise.

Examples

```
obj.createOperationAsConfigurableSubsystems('/PortInterface/csinterface')
```

See Also

"Importing an AUTOSAR Software Component" and "Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create file correlating tolerance information with signal names

Syntax

```
cgvCGV.createToleranceFile('file_name', signal_list,
   tolerance_list)
```

Description

`cgvCGV.createToleranceFile('file_name', signal_list, tolerance_list)` creates a MATLAB file, *`'file_name'`*, containing the tolerance specification for each output signal name in *`signal_list`*. Each signal name in the *`signal_list`* corresponds to the same location of each parameter name and value pair in the *`tolerance_list`*.

Input Arguments

`'file_name'`

Name for the file containing the tolerance specification for each signal. Use this file as input to the

`signal_list`

A cell array of strings, where each string is a signal name in the `dataset`. Use `cgv.CGV.getSavedSignals` to view the list of available signal names in the `dataset`. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for multiple signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)',...
'log_data.block_name.Data(:,3)',...
'log_data.block_name.Data(:,4)'};
```

To specify a global tolerance for all signals, include the reserved signal name, `'global_tolerance'`, in `signal_list`. Assign a global tolerance value in the associated `tolerance_list`. If `signal_list` contains other signals, their associated tolerance value overrides the global tolerance value. In this example, the global tolerance is a relative tolerance of `0.02`.

```
signal_list = {'global_tolerance',...
'log_data.block_name.Data(:,1)',...
'log_data.block_name.Data(:,2)'};

tolerance_list = {{'relative', 0.02},...
 {'relative', 0.015},{'absolute', 0.05}};
```

Note If a model component contains a space or newline character, MATLAB adds parantheses and a single quote to the name of the component. For example, if a substring of the signal name has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes in the `signal_list`. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

`tolerance_list`

Cell array of cell arrays. Each element of the outer cell array is a cell array containing a parameter name and value pair for the type of tolerance and its value. Possible parameter names are `'absolute' | 'relative' | 'function'`. There is a one-to-one mapping between each parameter name and value pair in the *`tolerance_list`* and a signal name in the *`signal_list`*. For example, a *`tolerance_list`* for a *`signal_list`* containing four signals might look like the following:

```
tolerance_list = {{'relative', 0.02},{'absolute', 0.06},...
 {'relative', 0.015},{'absolute', 0.05}};
```

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

Purpose

Display results of comparing configuration parameter values

Syntax

```
cfgObj.displayReport()
```

Description

`cfgObj.displayReport()` displays the results at the MATLAB Command Window of comparing the configuration parameter values for the model with the values that the object recommends. *`cfgObj`* is a handle to a `cgv.Config` object.

Tutorials

- "How To Verify a SIL or PIL Configuration"

How To

- "Configuring a PIL Simulation"

enableCPP

Purpose

Enable C++ support for function entry in TFL table

Syntax

```
enableCPP(hEntry)
```

Arguments

hEntry

Handle to a TFL function entry previously returned by *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from `RTW.TflCFunctionEntry`.

Description

The `enableCPP` function enables C++ support for a function entry in a TFL table. This allows you to specify a C++ name space for the implementation function defined in the entry (see the `setNameSpace` function).

Note When you register a TFL containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the TFL registry entry. For more information, see "Registering Target Function Libraries".

Example

In the following example, the `enableCPP` function is used to enable C++ support, and then the `setNameSpace` function is called to set the name space for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                'Key',                      'sin', ...
                                'Priority',                 100, ...
                                'ImplementationName',       'sin', ...
                                'ImplementationHeaderFile', 'cmath' );
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

See Also

`registerCPPFunctionEntry`, `setNameSpace`

"Example: Mapping Math Functions to Target-Specific Implementations" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

rtw.codegenObjectives.Objective.excludeCheck

Purpose

Exclude checks

Syntax

```
excludeCheck(obj, checkID)
```

Description

`excludeCheck(obj, checkID)` excludes a check from the Code Generation Advisor when a user specifies the objective. When a user selects multiple objectives, if the user specifies an additional objective that includes this check as a higher priority objective, the Code Generation Advisor displays this check.

Input Arguments

obj	Handle to a code generation objective object previously created.
checkID	Unique identifier of the check that you exclude from the new objective.

Examples

Exclude the **Identify questionable code instrumentation (data I/O)** check from the objective.

```
excludeCheck(obj, 'Identify questionable code instrumentation (data I/O)');
```

See Also

`Simulink.ModelAdvisor`

How To

- "Creating Custom Objectives"
- "About IDs"

Purpose

Get argument category for Simulink model port from model-specific C++ encapsulation interface

Syntax

```
category = getArgCategory(obj, portName)
```

Description

`category = getArgCategory(obj, portName)` gets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

category	String specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — for the specified Simulink model port.

Alternatives

To view argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get argument category for Simulink model port from model-specific C function prototype

Syntax

```
category = getArgCategory(obj, portName)
```

Description

`category = getArgCategory(obj, portName)` gets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

category	String specifying the argument category, `'Value'` or `'Pointer'`, for the specified Simulink model port.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument categories. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.ModelCPPArgsClass.getArgName

Purpose

Get argument name for Simulink model port from model-specific C++ encapsulation interface

Syntax

```
argName = getArgName(obj, portName)
```

Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

argName	String specifying the argument name for the specified Simulink model port.

Alternatives

To view argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.getArgName

Purpose

Get argument name for Simulink model port from model-specific C function prototype

Syntax

```
argName = getArgName(obj, portName)
```

Description

`argName = getArgName(obj, portName)` gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

argName	String specifying the argument name for the specified Simulink model port.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument names. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Get argument position for Simulink model port from model-specific C++ encapsulation interface

Syntax

```
position = getArgPosition(obj, portName)
```

Description

`position = getArgPosition(obj, portName)` gets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

position	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. If there is no argument for the specified port, the function returns 0.

Alternatives

To view argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get argument position for Simulink model port from model-specific C function prototype

Syntax

```
position = getArgPosition(obj, portName)
```

Description

`position = getArgPosition(obj, portName)` gets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

position	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. If no argument is found for the specified port, the function returns 0.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument positions. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.ModelCPPArgsClass.getArgQualifier

Purpose

Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface

Syntax

```
qualifier = getArgQualifier(obj, portName)
```

Description

`qualifier = getArgQualifier(obj, portName)` gets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

qualifier	String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — for the specified Simulink model port.

Alternatives

To view argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers. For more information, see "Configuring the Step Method

for Your Model Class” in the Real-Time Workshop Embedded Coder documentation.

See Also

“Configuring C++ Encapsulation Interfaces Programmatically” in the Real-Time Workshop Embedded Coder documentation

“Sample Script for Configuring the Step Method for a Model Class” in the Real-Time Workshop Embedded Coder documentation

“Controlling Generation of Encapsulated C++ Model Interfaces” in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.getArgQualifier

Purpose

Get argument type qualifier for Simulink model port from model-specific C function prototype

Syntax

```
qualifier = getArgQualifier(obj, portName)
```

Description

`qualifier = getArgQualifier(obj, portName)` gets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — of the argument corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

qualifier	String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — for the specified Simulink model port.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument qualifiers. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Get calibration component names

Syntax

```
calibrationComponentNames = importerObj.getCalibrationComponen
   tNames
```

Description

`calibrationComponentNames = importerObj.getCalibrationComponentNames` returns the list of calibration component names found in the XML files associated with the `arxml.importer` object, *`importerObj`*.

Output Arguments

`calibration ComponentNames`	Cell array of strings in which each element is the absolute short name path of the corresponding calibration parameter component : `'/root_package_name[/sub_package_name]/component_short_name'`

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelCPPClass.getClassName

Purpose

Get class name from model-specific C++ encapsulation interface

Syntax

```
clsName = getClassName(obj)
```

Description

`clsName = getClassName(obj)` gets the name of the class described by the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.

Output Arguments

clsName	A string specifying the name of the class described by the specified model-specific C++ encapsulation interface.

Alternatives

To view the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, which displays the model class name and allows you to display and configure the step method for your model class. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get XML component name

Syntax

```
componentName = autosarInterfaceObj.getComponentName
```

Description

`componentName = autosarInterfaceObj.getComponentName` gets the XML component name of the model-specific `RTW.AutosarInterface` object defined by *`autosarInterfaceObj`*.

Output Arguments

`componentName`	Name of XML component object defined by *`autosarInterfaceObj`*.

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

arxml.importer.getComponentNames

Purpose

Get atomic software component names

Syntax

```
componentNames = importerObj.getComponentNames
```

Description

`componentNames = importerObj.getComponentNames` returns the list of atomic software component names in the XML file associated with the `arxml.importer` object, *`importerObj`*.

Note `getComponentNames` finds only the atomic software component defined in the XML file specified when constructing the `arxml.importer` object or the XML file specified by the method `setFile`. All atomic software components described in the XML file dependencies are ignored.

Output Arguments

`componentNames`	Cell array of strings in which each element is the absolute short name path of the corresponding atomic software component :

```
'/root_package_name[/sub_package_name]/component_short_name'
```

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get XML data type package name

Syntax

```
dataTypePackageName = getDataTypePackageName(autosarInterfaceObj)
```

Description

`dataTypePackageName = getDataTypePackageName(autosarInterfaceObj)` gets the XML data type package name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Output Arguments

`dataTypePackageName`	Name of the data type package specified by *`autosarInterfaceObj`*.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getDefaultConf

Purpose

Get default configuration

Syntax

```
autosarInterfaceObj.getDefaultConf
```

Description

autosarInterfaceObj`.getDefaultConf` gets the model's default configuration for *autosarInterfaceObj*, using information from the model to which *autosarInterfaceObj* is attached.

autosarInterfaceObj is a model-specific `RTW.AutosarInterface` object. You must attach the object to a model using `attachToModel` before calling `getDefaultConf`.

When you initially invoke `getDefaultConf` (or the GUI button equivalent, **Get Default Configuration** in the Model Interface dialog), the runnable names, XML properties, and I/O configuration are initialized. If you invoke the command (or click the button) again, only the I/O configurations are reset to default values.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get default configuration information for model-specific C++ encapsulation interface from Simulink model

Syntax

```
getDefaultConf(obj)
```

Description

`getDefaultConf(obj)` initializes the specified model-specific C++ encapsulation interface to a default configuration, based on information from the ERT-based Simulink model to which the interface is attached. On the first invocation, class and step method names and step method properties are set to default values. On subsequent invocations, only step method properties are reset to default values.

Before calling this function, you must call `attachToModel`, to attach the C++ encapsulation interface to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.ModelCPPVoidClass`.

Alternatives

To view C++ encapsulation interface default configuration information in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display default configuration information. In the `void-void step method` view, you can see the default configuration information without clicking a button. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get default configuration information for model-specific C function prototype from Simulink model

Syntax

```
getDefaultConf(obj)
```

Description

`getDefaultConf(obj)` invokes the specified model-specific C function prototype to initialize the properties and the step function name of the function argument to a default configuration based on information from the ERT-based Simulink model to which it is attached. If you invoke the command again, only the properties of the function argument are reset to default values.

Before calling this function, you must call `attachToModel`, to attach the function prototype to a loaded model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype`.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get the default configuration. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

arxml.importer.getDependencies

Purpose

Get list of XML dependency files

Syntax

```
Dependencies = importerObj.getDependencies()
```

Description

`Dependencies = importerObj.getDependencies()` returns the list of XML dependency files associated with the `arxml.importer` object, *`importerObj`*.

Output Arguments

`Dependencies`	Cell array of strings.

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get runnable execution period

Syntax

```
EP = autosarInterfaceObj.getExecutionPeriod
```

Description

`EP = autosarInterfaceObj.getExecutionPeriod` gets the execution period in the configuration.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Output Arguments

`EP`	Execution period in the configuration

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

arxml.importer.getFile

Purpose

Return XML file name for `arxml.importer` object

Syntax

```
filename = importerObj.getFile
```

Description

`filename = importerObj.getFile` returns the name of the XML file associated with the `arxml.importer` object, *`importerObj`*.

Output Arguments

`filename`	XML file name

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get function name from model-specific C function prototype

Syntax

```
fcnName = getFunctionName(obj, fcnType)
```

Description

`fcnName = getFunctionName(obj, fcnType)` gets the name of the step or initialize function described by the specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
fcnType	Optional string specifying which function name to get. Valid strings are `'step'` and `'init'`. If *fcnType* is not specified, gets the step function name.

Output Arguments

fcnName	A string specifying the name of the function described by the specified model-specific C function prototype.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get function names. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.AutosarInterface.getImplementationName

Purpose

Get XML implementation name

Syntax

```
implementationName = autosarInterfaceObj.getImplementationName
```

Description

`implementationName = autosarInterfaceObj.getImplementationName` gets the XML implementation name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get initial event name

Syntax

```
initEventName = autosarInterfaceObj.getInitEventName
```

Description

`initEventName = autosarInterfaceObj.getInitEventName` gets the initial event name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Output Arguments

`initEventName`	Name of the initial event specified by *`autosarInterfaceObj`*.

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getInitRunnableName

Purpose

Get initial runnable name

Syntax

```
initRunnableName = autosarInterfaceObj.getInitRunnableName
```

Description

`initRunnableName = autosarInterfaceObj.getInitRunnableName` gets the initial runnable name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Output Arguments

`initRunnableName`	Name of the initial runnable specified by *`autosarInterfaceObj`*.

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get XML interface package name

Syntax

```
interfacePkgName = autosarInterfaceObj.getInterfacePackageName
```

Description

`interfacePkgName = autosarInterfaceObj.getInterfacePackageName` gets the XML interface package name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Output Arguments

`interfacePkgName`	Name of the interface package specified by *`autosarInterfaceObj`*.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getInternalBehaviorName

Purpose

Get XML internal behavior name

Syntax

```
internalBehaviorName = autosarInterfaceObj.getInternalBehavior
   Name
```

Description

`internalBehaviorName = autosarInterfaceObj.getInternalBehaviorName` gets the XML internal behavior name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Output Arguments

`internalBehavior Name`	Name of the internal behavior specified by *`autosarInterfaceObj`*.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get I/O AUTOSAR port name

Syntax

```
ioAutosarName = autosarInterfaceObj.getIOAutosarPortName(portN
   ame)
```

Description

ioAutosarName = *autosarInterfaceObj*.getIOAutosarPortName(*portName*) gets the I/O AUTOSAR port name in the configuration for the port corresponding to *portName*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of inport/outport name (string).

Output Arguments

ioAutosarName	AUTOSAR port name of *portName*

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getIODataAccessMode

Purpose

Get I/O data access mode

Syntax

```
dataAccessMode = autosarInterfaceObj.getIODataAccessMode(portName)
```

Description

`dataAccessMode = autosarInterfaceObj.getIODataAccessMode(portName)` gets the data access mode of the I/O corresponding to *`portName`*, for `autosarInterfaceObj`, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`portName`	Name of inport/outport (string).

Output Arguments

`dataAccessMode`	Data access mode of the given port. Can be one of the following: • `'ImplicitSend'` • `'ImplicitReceive'` • `'ExplicitSend'` • `'ExplicitReceive'`

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get I/O data element name

Syntax

```
ioDataElement = autosarInterfaceObj.getIODataElement(portName
    )
```

Description

`ioDataElement = autosarInterfaceObj.getIODataElement(portName)` gets the I/O data element name in the configuration for the port corresponding to *portName*.

autosarInterfaceObj is a model-specific `RTW.AutosarInterface` object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of inport/outport (string).

Output Arguments

ioDataElement	Data element of the given port (string).

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getIOErrorStatusReceiver

Purpose

Get receiver port name

Syntax

```
ESR = autosarInterfaceObj.getIOErrorStatusReceiver(PortName)
```

Description

`ESR = autosarInterfaceObj.getIOErrorStatusReceiver(PortName)` gets the receiver port name in the configuration for the port corresponding to `PortName` .

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`	Name of inport/outport (string).

Output Arguments

`ESR`	Name of receiver port for `PortName`.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get I/O interface name

Syntax

```
ioInterfaceName = autosarInterfaceObj.getIOInterfaceName(portName)
```

Description

ioInterfaceName = *autosarInterfaceObj*.getIOInterfaceName(*portName*)
gets the I/O interface name in the configuration for the port corresponding to *portName*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of the inport/outport (string).

Output Arguments

ioInterfaceName	Name of the I/O interface for *portName*.

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getIOPortNumber

Purpose

Get I/O AUTOSAR port number

Syntax

```
IOPortNumber= autosarInterfaceObj.getIOPortNumber(PortName)
```

Description

`IOPortNumber= autosarInterfaceObj.getIOPortNumber(PortName)` gets the I/O AUTOSAR port number in the configuration for the port corresponding to *`PortName`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`	Name of the inport/output (string).

Output Arguments

`IOPortNumber`	Port number of *`PortName`*.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get port I/O service interface

Syntax

```
SI = autosarInterfaceObj.getIOServiceInterface(PortName)
```

Description

`SI = autosarInterfaceObj.getIOServiceInterface(PortName)` gets the I/O service interface in the configuration for the port corresponding to `PortName`.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`	Name of the inport/outport (string).

Output Arguments

`SI`	I/O service interface of `PortName`.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getIOServiceName

Purpose

Get port I/O service name

Syntax

```
SN = autosarInterfaceObj.getIOServiceName(PortName)
```

Description

`SN = autosarInterfaceObj.getIOServiceName(PortName)` gets the I/O service name in the configuration for the port corresponding to `PortName`.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`	Name of the inport/outport (string).

Output Arguments

`SN`	Name of I/O service for `PortName`.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get port I/O service operation

Syntax

```
SO = autosarInterfaceObj.getIOServiceOperation(PortName)
```

Description

`SO = autosarInterfaceObj.getIOServiceOperation(PortName)` gets the I/O service operation in the configuration for the port corresponding to `PortName`.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`	Inport/outport name (string).

Output Arguments

`SO`	I/O service operation of *`PortName`*.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getIsServerOperation

Purpose

Determine whether server is specified

Syntax

```
isServerOperation = autosarInterfaceObj.getIsServerOperation
```

Description

isServerOperation = *autosarInterfaceObj*.getIsServerOperation returns the value of the property *'isServerOperation'* in *autosarInterfaceObj*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Output Arguments

isServerOperation	True or false. If true, a server is specified in *autosarInterfaceObj*.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get number of step method arguments from model-specific C++ encapsulation interface

Syntax

```
num = getNumArgs(obj)
```

Description

`num = getNumArgs(obj)` gets the number of arguments for the step method described by the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.

Output Arguments

num	An integer specifying the number of step method arguments.

Alternatives

To view the number of step method arguments in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the step method arguments. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get number of function arguments from model-specific C function prototype

Syntax

```
num = getNumArgs(obj)
```

Description

`num = getNumArgs(obj)` gets the number of function arguments for the function described by the specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.

Output Arguments

num	An integer specifying the number of function arguments.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get arguments. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

cgv.CGV.getOutputData

Purpose

Get output data

Syntax

```
out = cgvObj.getOutputData(InputIndex)
```

Description

`out = cgvObj.getOutputData(InputIndex)` is the method that you use to retrieve the output data that the object creates during execution of the model. `out` is the output data that the object returns. `cgvObj` is a handle to a `cgv.CGV` object. `InputIndex` is a unique numeric identifier that specifies which output data to retrieve. The `InputIndex` is associated with specific input data.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

Purpose

Get periodic event name

Syntax

```
periodicEventName = autosarInterfaceObj.getPeriodicEventName
```

Description

`periodicEventName = autosarInterfaceObj.getPeriodicEventName` gets the periodic event name specified by the model-specific `RTW.AutosarInterface` object, *`autosarInterfaceObj`*.

Output Arguments

`periodicEventName`	Name of the periodic event specified by *`autosarInterfaceObj`*

Examples

For multiple runnables, use the Children property to access each individual runnable after building or GUI update, for example:

```
autosarInterfaceObj.Children(1).getPeriodicEventName()
```

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getPeriodicRunnableName

Purpose

Get periodic runnable name

Syntax

```
periodicRunnableName = autosarInterfaceObj.getPeriodicRunnable
   Name
```

Description

periodicRunnableName = *autosarInterfaceObj*.getPeriodicRunnableName gets the name of the periodic runnable specified in autosarInterfaceObj, a model-specific RTW.AutosarInterface object.

Output Arguments

periodicRunnable Name	Name of the periodic runnable specified by *autosarInterfaceObj*.

Examples

For multiple runnables, use the Children property to access each individual runnable after building or GUI update, for example:

```
autosarInterfaceObj.Children(1).getPeriodicRunnableName()
```

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get port default configuration

Syntax

```
[autosarPort, interfaceName, dataElement,
   dataAccessMode]=autosarInterfaceObj.getPortDefaultConf(port
   H)
```

Description

`[autosarPort, interfaceName, dataElement, dataAccessMode]=autosarInterfaceObj.getPortDefaultConf(portH)` gets the port's default configuration for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`portH`	Port of *`autosarInterfaceObj`*.

Output Arguments

`autosarPort`	AUTOSAR port name for *`portH`*.
`interfaceName`	Name I/O interface for *`portH`* (string).
`dataElement`	Data element of *`portH`* (string).
`dataAccessMode`	Data access mode of *`portH`* , and is one of the following: • `'ImplicitSend'` • `'ImplicitReceive'` • `'ExplicitSend'` • `'ExplicitReceive'`

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.getPreview

Purpose

Get model-specific C function prototype code preview

Syntax

```
preview = getPreview(obj, fcnType)
```

Description

`preview = getPreview(obj, fcnType)` gets the model-specific C function prototype code preview.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.getFunctionSpecification(modelName)`.
fcnType	Optional. String specifying which function to preview. Valid strings are `'step'` and `'init'`. If *fcnType* is not specified, previews the step function.

Output Arguments

preview	String specifying the function prototype for the step or initialization function.

Alternatives

Use the **Step function preview** subpane in the Model Interface dialog box to preview how your step function prototype is interpreted in generated code. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Return results of comparing configuration parameter values

Syntax

```
rpt_data = cfgObj.getReportData()
```

Description

`rpt_data = cfgObj.getReportData()` compares the original configuration parameter values with the values that the object recommends. *`cfgObj`* is a handle to a `cgv.Config` object. Returns a cell array of strings with the model, parameter, previous value, and recommended or new value.

Tutorials

- "How To Verify a SIL or PIL Configuration"

How To

- "Configuring a PIL Simulation"

cgv.CGV.getSavedSignals

Purpose

Display list of signal names to command line

Syntax

```
signal_list = cgvCGV.getSavedSignals(simulation_data)
```

Description

`signal_list = cgvCGV.getSavedSignals(simulation_data)` returns a cell array, *`signal_list`*, of all output signal names of all data elements from the input data set, *`simulation_data`*. *`simulation_data`* is the output data stored in the CGV object when you execute the model.

Tips

- After executing your model, use the `cgv.CGV.getOutputData` function to get the output data used as the input argument to the `cgv.CGV.getSavedSignals` function.
- Use names from the output signal list at the command line or as input arguments to other CGV functions, for example, `cgv.CGV.createToleranceFile`, `cgv.CGV.compare`, and `cgv.CGV.plot`.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

Purpose

Get name of server interface

Syntax

```
serverInterfaceName = autosarInterfaceObj.getServerInterfaceNa
   me
```

Description

`serverInterfaceName = autosarInterfaceObj.getServerInterfaceName` returns the name of the server interface specified in *`autosarInterfaceObj`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Output Arguments

`serverInterfaceName`	Name of the server interface in *`autosarInterfaceObj`*.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getServerOperationPrototype

Purpose

Get server operation prototype

Syntax

```
operation_prototype = autosarInterfaceObj.getServerOperationPr
   ototype
```

Description

operation_prototype = *autosarInterfaceObj*.getServerOperationPrototype returns the server operation prototype in *autosarInterfaceObj*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Output Arguments

operation_prototype

String with names of prototype and arguments:

operation_name(*dir1 datatype1 arg1*, *dir2 datatype2 arg2*, ..., *dirN datatypeN argN*, ...)

- *operation_name* — Name of the operation
- *dirN* — Either IN or OUT, which indicates whether data is passed in or out of the function.
- *datatypeN* — Data type, which can be an AUTOSAR basic data type or record, Simulink data type, or array.
- *argN* — Name of the argument

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get server port name

Syntax

```
serverPortName = autosarInterfaceObj.getServerPortName
```

Description

`serverPortName = autosarInterfaceObj.getServerPortName` returns the server port name of the model-specific `RTW.AutosarInterface` object defined by *`autosarInterfaceObj`*.

Output Arguments

`serverPortName`	Name of the server port defined by *`autosarInterfaceObj`*.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.getServerType

Purpose

Determine server type

Syntax

```
serverType = autosarInterfaceObj.getServerType
```

Description

`serverType = autosarInterfaceObj.getServerType` determines the type of the server in *`autosarInterfaceObj`*, that is, whether it is application software or Basic software.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Output Arguments

`serverType`	Either `'Application software'` or `'Basic software'`.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get step method name from model-specific C++ encapsulation interface

Syntax

```
fcnName = getStepMethodNameName(obj)
```

Description

`fcnName = getStepMethodNameName(obj)` gets the name of the step method described by the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.

Output Arguments

fcnName	A string specifying the name of the step method described by the specified model-specific C++ encapsulation interface.

Alternatives

To view the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, which displays the step method name and allows you to display and configure the step method for your model class. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create TFL argument based on specified name and built-in data type

Syntax

```
arg = getTflArgFromString(hTable, name, datatype)
```

Arguments

hTable
: Handle to a TFL table previously returned by *hTable* = `RTW.TflTable`.

name
: String specifying the name to use for the TFL argument, for example, `'y1'`.

datatype
: String specifying the built-in data type to use for the TFL argument, among the following:`'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, `'single'`, `'double'`, or `'boolean'`.

Returns

Handle to the created TFL argument, which can be specified to the `addConceptualArg` function. See the example below.

Description

The `getTflArgFromString` function creates a TFL argument that is based on a specified name and built-in data type.

Note The `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, you must change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property. See the example below.

Example

In the following example, `getTflArgFromString` is used to create an `int16` output argument named `y1`, which is then added as a conceptual argument for a TFL table entry.

```
hLib = RTW.TflTable;
op_entry = RTW.TflCOperationEntry;
```

```
.
.
.
arg = hLib.getTflArgFromString('y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );
```

See Also

`addConceptualArg`

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Initialization entry point in generated code for ERT-based Simulink model

Syntax

```
void model_initialize(void)
void model_initialize(boolean_T firstTime)
```

Argument

firstTime

The Real-Time Workshop Embedded Coder software generates the *firstTime* argument to *model*_initialize only if both of the following conditions are true:

- Your selected target supports *firstTime* argument control — that is, target configuration parameter ERTFirstTimeCompliant is set to on. (ERT targets supplied by The MathWorks™ support *firstTime* argument control.)
- The IncludeERTFirstTime model configuration parameter, which is off by default, is set to on.

The *firstTime* argument specifies value 0 (FALSE) or 1 (TRUE). If *firstTime* equals 1, *model*_initialize initializes rtModel and other data structures private to the model. If *firstTime* equals 0, *model*_initialize resets the model's states, but does not initialize other data structures. Call *model*_initialize with *firstTime* set to 0 to reset the model's states at a time greater than start time.

Note In a future release, the Real-Time Workshop Embedded Coder software will discontinue use of the *firstTime* argument in a model's generated *model*_initialize function.

Description

The *model*_initialize function contains all model initialization code.

The generated code for a Simulink model calls *model*_initialize once, at the beginning of model execution. If your selected target supports

firstTime argument control and IncludeERTFirstTime is set to on, the generated code passes in *firstTime* as 1 (TRUE).

See Also

model_SetEventsForThisBaseStep, model_step, model_terminate

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

Command Line Information for IncludeERTFirstTime and ERTFirstTimeCompliant in the Real-Time Workshop Embedded Coder reference documentation

Purpose

Set event flags for multirate, multitasking operation before calling *model*_step for ERT-based Simulink model — not generated as of Version 5.1 (R2008a)

Syntax

```
void model_SetEventsForThisBaseStep(boolean_T *eventFlags)
void model_SetEventsForThisBaseStep(boolean_T *eventFlags,
RT_MODEL_model *model_M)
```

Arguments

eventFlags
: Pointer to the model's event flags array.

*model*_M
: Pointer to the real-time model object. The Real-Time Workshop Embedded Coder software generates this argument only if **Generate reusable code** is on.

Description

Versions of the Real-Time Workshop Embedded Coder software prior to Version 5.1 (R2008a) generate the *model*_SetEventsForThisBaseStep function for multirate, multitasking models. The function maintains model event flags that determine which subrate tasks need to run on a given base rate time step. In a multirate, multitasking application, the program code must call *model*_SetEventsForThisBaseStep before calling the *model*_step function.

Note The macro MODEL_SETEVENTS, defined in the static ert_main.c module, provides a way to call *model*_SetEventsForThisBaseStep from a static main program.

Note Real-Time Workshop Embedded Coder no longer generates this function and you should avoid using it. The model event flags are now maintained by code in a model's generated example main program (ert_main.c). For more information, see "Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets".

See Also

`model_initialize`, `model_step`, `model_terminate`

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

Purpose

Step routine entry point in generated code for ERT-based Simulink model

Syntax

```
void model_step(void)
void model_step(int_T tid)
void model_stepN(void)
```

Arguments

tid

Task identifier. The Real-Time Workshop Embedded Coder software generates this argument only for multirate, single-tasking models.

Calling Interfaces

The *model*_step default function prototype varies depending on the number of rates in the model and the solver mode, as shown below:

Rates/Solver Mode	Function Prototype
Single-rate/SingleTasking	`void model_step(void);`
Multirate/SingleTasking	`void model_step(int_T tid);`
Multirate/MultiTasking (rate grouping)	`void model_stepN (void);` (*N* is a task identifier)

If you generate reusable, reentrant code for an ERT-based model using the **Generate reusable code** option, the generated code passes the model's root-level inputs and outputs, block states, parameters, and external outputs to *model*_step using a function prototype that generally resembles the following:

```
void model_step(inport_args, outport_args, BlockIO_arg,
DWork_arg, RT_model_arg);
```

The manner in which the inport and outport arguments are passed is determined by the setting of the **Pass root-level I/O as** parameter, which appears on the **Interface** pane of the Configuration Parameters dialog box only if **Generate reusable code** is selected.

For greater control over the *model*_step function prototype, you can use the **Configure Model Functions** button on the **Interface** pane to launch a Model Interface dialog box (see "Configuring Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation). Based on the **Function specification** value you specify for your *model*_step function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications. For more information about controlling the *model*_step function prototype, see the sections "Configuring the Target Hardware Environment" and "Controlling Generation of Function Prototypes" in the Real-Time Workshop Embedded Coder documentation.

Description

The Real-Time Workshop Embedded Coder software generates the *model*_step function for a Simulink model when the **Single output/update function** configuration option is selected (the default) in the Configuration Parameters dialog box. *model*_step contains the output and update code for all blocks in the model.

*model*_step is designed to be called at interrupt level from rt_OneStep, which is assumed to be invoked as a timer ISR. rt_OneStep calls *model*_step to execute processing for one clock period of the model. See "rt_OneStep and Scheduling Considerations" in the Real-Time Workshop Embedded Coder documentation for a description of how calls to *model*_step are generated and scheduled.

Note If the **Single output/update function** configuration option is not selected, the Real-Time Workshop Embedded Coder software generates the following model entry point functions in place of *model*_step:

- *model*_output: Contains the output code for all blocks in the model
- *model*_update: Contain the update code for all blocks in the model

The *model*_step function computes the current value of all blocks. If logging is enabled, *model*_step updates logging variables. If the model's stop time is finite, *model*_step signals the end of execution when the current time equals the stop time.

In cases where a *tid* is passed in, the caller (rt_OneStep) assigns each task a *tid*, and *model*_step uses the *tid* argument to determine which blocks have a sample hit (and, therefore, should execute).

Under any of the following conditions, *model*_step does not check the current time against the stop time:

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

Therefore, if any of these conditions are true, the program runs indefinitely.

See Also

model_initialize, model_SetEventsForThisBaseStep, model_terminate

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

model_terminate

Purpose

Termination entry point in generated code for ERT-based Simulink model

Syntax

```
void model_terminate(void)
```

Description

The Real-Time Workshop Embedded Coder software generates the *model*_terminate function for a Simulink model when the **Terminate function required** configuration option is selected (the default) in the Configuration Parameters dialog box. *model*_terminate contains all model termination code and should be called as part of system shutdown.

When *model*_terminate is called, blocks that have a terminate function execute their terminate code. If logging is enabled, *model*_terminate ends data logging.

The *model*_terminate function should be called only once.

If your application runs indefinitely, you do not need the *model*_terminate function. To suppress the function, clear the **Terminate function required** configuration option in the Configuration Parameters dialog box.

See Also

model_initialize, model_SetEventsForThisBaseStep, model_step

"Model Entry Points" in the Real-Time Workshop Embedded Coder documentation

rtw.codegenObjectives.Objective.modifyInheritedParam

Purpose

Modify inherited parameter values

Syntax

```
modifyInheritedParam(obj, paramName, value)
```

Description

`modifyInheritedParam(obj, paramName, value)` changes the value of an inherited parameter that the Code Generation Advisor verifies in **Check model configuration settings against code generation objectives**. Use this method when you create a new objective from an existing objective.

Input Arguments

obj	Handle to a code generation objective object previously created.
paramName	Parameter that you modify in the objective.
value	Value of the parameter.

Examples

Change the value of `Inlineparameters` to `off` in the objective.

```
modifyInheritedParam(obj, 'InlineParams', 'off');
```

See Also

`get_param`

How To

- "Creating Custom Objectives"
- "Parameter Command-Line Information Summary"

cgv.CGV.plot

Purpose

Create plot for signal or multiple signals

Syntax

```
[signal_names, signal_figures] = cgv.CGV.plot(dataset)
[signal_names, signal_figures] = cgv.CGV.plot(dataset,
   'Signals', signal_list)
```

Description

`[signal_names, signal_figures] = cgv.CGV.plot(dataset)` create a plot for each signal in the *`dataset`*.

`[signal_names, signal_figures] = cgv.CGV.plot(dataset,'Signals', signal_list)` create a plot for each signal in the value of `'signals'` and return the names and figure handles for the given signal names.

Input Arguments

`dataset`

Output data from a model. After running the model, use the `cgv.CGV.getOutputData` function to get the data. The `cgv.CGV.getOutputData` function returns a cell array of all output signal names.

`'Signals', signal_list`

Parameter/value argument pair specifying the signal or signals to plot. The value for this parameter can be an individual signal name, or a cell array of strings, where each string is a signal name in the *`dataset`*. Use `cgv.CGV.getSavedSignals` to view the list of available signal names in the *`dataset`*. The syntax for an individual signal name is:

```
signal_list = {'log_data.subsystem_name.Data(:,1)'}
```

The syntax for a list of signal names is:

```
signal_list = {'log_data.block_name.Data(:,1)',...
          'log_data.block_name.Data(:,2)',...
                      'log_data.block_name.Data(:,3)',...
                      'log_data.block_name.Data(:,4)'};
```

If a component of your model contains a space or newline character, MATLAB adds parantheses and a single quote to the name of the component. For example, if a section of the signal has a space, `'block name'`, MATLAB displays the signal name as:

```
log_data.('block name').Data(:,1)
```

To use the signal name as input to a CGV function, `'block name'` must have two single quotes. For example:

```
signal_list = {'log_data.(''block name'').Data(:,1)'}
```

Output Arguments

Depending on the data, any of the following parameters might be empty:

`signal_names`

Cell array of signal names

`signal_figures`

Array of figure handles for signals

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

rtw.codegenObjectives.Objective.register

Purpose

Register objective

Syntax

```
register(obj)
```

Description

`register(obj)` registers *`obj`* Register and add *`obj`* to the end of the list of available objectives that you can use with the Code Generation Advisor.

Input Arguments

`obj`	Handle to a code generation objective object previously created.

Examples

Register the objective:

```
register(obj);
```

See Also

`DAStudio.CustomizationManager.ObjectiveCustomizer`

How To

- "Creating Custom Objectives"
- "Registering Customizations"

Purpose

Create TFL function entry based on specified parameters and register in TFL table

Syntax

```
entry = registerCFunctionEntry(hTable, priority,
                               numInputs, functionName,
                               inputType, implementationName,
                               outputType, headerFile,
                               genCallback, genFileName)
```

Arguments

hTable

Handle to a TFL table previously returned by `hTable = RTW.TflTable`.

priority

Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

numInputs

Positive integer specifying the number of input arguments.

functionName

String specifying the name of the function to be replaced. The name must match one of the functions supported for replacement:

Math Functions			
abs	cos	log	saturate
acos	cosh	log10	sign
acosh	exactrSqrt	max	sin
asin	exp	min	sinh
asinh	fix	mod/fmod	sqrt
atan	floor	pow	tan

atan2	hypot	rem	tanh
atanh	ldexp	round	
ceil	ln	rSqrt	
Copy Utility Function			
memcpy			
Nonfinite Support Utility Functions			
getInf	getMinusInf	getNaN	

inputType
: String specifying the data type of the input arguments, for example, `'double'`. (This function requires that all input arguments are of the same type.)

implementationName
: String specifying the name of your implementation. For example, if *functionName* is `'sqrt'`, *implementationName* can be `'sqrt'` or a different name of your choosing.

outputType
: String specifying the data type of the return argument, for example, `'double'`.

headerFile
: String specifying the header file in which the implementation function is declared, for example, `'<math.h>'`.

genCallback
: String specifying `''` or `'RTW.copyFileToBuildDir'`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation.

genFileName
: String specifying `''`. (This argument is for use only by MathWorks™ developers.)

Returns

Handle to the created TFL function entry. Specifying the return argument in the `registerCFunctionEntry` function call is optional.

Description

The `registerCFunctionEntry` function provides a quick way to create and register a TFL function entry. This function can be used only if your TFL function entry meets the following conditions:

- All input arguments are of the same type.
- All input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `un`
  - For return argument, `y1`

Example

In the following example, the `registerCFunctionEntry` function is used to create a function entry for `sqrt` in a TFL table.

```
hLib = RTW.TflTable;

hLib.registerCFunctionEntry(100, 1, 'sqrt', 'double', 'sqrt', ...
                            'double', '<math.h>', '', '');
```

See Also

`registerCPromotableMacroEntry`

"Alternative Method for Creating Function Entries" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

registerCPPFunctionEntry

Purpose

Create TFL C++ function entry based on specified parameters and register in TFL table

Syntax

```
entry = registerCPPFunctionEntry(hTable, priority,
                                 numInputs, functionName,
                                 inputType, implementationName,
                                 outputType, headerFile,
                                 genCallback, genFileName,
                                 nameSpace)
```

Arguments

hTable
: Handle to a TFL table previously returned by `hTable = RTW.TflTable`.

priority
: Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

numInputs
: Positive integer specifying the number of input arguments.

functionName
: String specifying the name of the function to be replaced. The name must match one of the functions supported for replacement:

Math Functions			
abs	cos	log	saturate
acos	cosh	log10	sign
acosh	exactrSqrt	max	sin
asin	exp	min	sinh
asinh	fix	mod/fmod	sqrt

atan	floor	pow	tan
atan2	hypot	rem	tanh
atanh	ldexp	round	
ceil	ln	rSqrt	
Copy Utility Function			
memcpy			
Nonfinite Support Utility Functions			
getInf	getMinusInf	getNaN	

inputType
: String specifying the data type of the input arguments, for example, `'double'`. (This function requires that all input arguments are of the same type.)

implementationName
: String specifying the name of your implementation. For example, if *functionName* is `'sqrt'`, *implementationName* can be `'sqrt'` or a different name of your choosing.

outputType
: String specifying the data type of the return argument, for example, `'double'`.

headerFile
: String specifying the header file in which the implementation function is declared, for example, `'<math.h>'`.

genCallback
: String specifying `''` or `'RTW.copyFileToBuildDir'`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation.

genFileName
String specifying `''`. (This argument is for use only by MathWorks developers.)

nameSpace
String specifying the C++ name space in which the implementation function is defined. If this function entry is matched, the software emits the name space in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`). If you specify `''`, the software does not emit a name space designation in the generated code.

Returns

Handle to the created TFL C++ function entry. Specifying the return argument in the `registerCPPFunctionEntry` function call is optional.

Description

The `registerCPPFunctionEntry` function provides a quick way to create and register a TFL C++ function entry. This function can be used only if your TFL C++ function entry meets the following conditions:

- All input arguments are of the same type.
- All input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

Note When you register a TFL containing C++ function entries, you must specify the value `{'C++'}` for the `LanguageConstraint` property of the TFL registry entry. For more information, see "Registering Target Function Libraries".

Example

In the following example, the `registerCPPFunctionEntry` function is used to create a C++ function entry for `sin` in a TFL table.

```
hLib = RTW.TflTable;
```

```
hLib.registerCPPFunctionEntry(100, 1, 'sin', 'single', 'sin', ...
                              'single', 'cmath', '', '', 'std');
```

See Also

`enableCPP`, `setNameSpace`

"Alternative Method for Creating Function Entries" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

registerCPromotableMacroEntry

Purpose

Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)

Syntax

```
entry = registerCPromotableMacroEntry(hTable, priority,
                                      numInputs, functionName,
                                      inputType, implementationName,
                                      outputType, headerFile,
                                      genCallback, genFileName)
```

Arguments

`hTable`
: Handle to a TFL table previously returned by *`hTable`* = `RTW.TflTable`.

`priority`
: Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

`numInputs`
: Positive integer specifying the number of input arguments.

`functionName`
: String specifying the name of the function to be replaced. Specify `'abs'`. (This function should be used only for `abs` function replacement.)

`inputType`
: String specifying the data type of the input arguments, for example, `'double'`. (This function requires that all input arguments are of the same type.)

`implementationName`
: String specifying the name of your implementation. For example, assuming *`functionName`* is `'abs'`, *`implementationName`* can be `'abs'` or a different name of your choosing.

outputType
String specifying the data type of the return argument, for example, `'double'`.

headerFile
String specifying the header file in which the implementation function is declared, for example, `'<math.h>'`.

genCallback
String specifying `''` or `'RTW.copyFileToBuildDir'`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation.

genFileName
String specifying `''`. (This argument is for use only by MathWorks developers.)

Returns

Handle to the created TFL promotable macro entry. Specifying the return argument in the `registerCPromotableMacroEntry` function call is optional.

Description

The `registerCPromotableMacroEntry` function creates a TFL promotable macro entry based on specified parameters and registers the entry in the TFL table. A promotable macro entry will promote the output data type based on the target word size.

This function provides a quick way to create and register a TFL promotable macro entry. This function can be used only if your TFL function entry meets the following conditions:

- All input arguments are of the same type.
- All input argument names and the return argument name follow the default Simulink naming convention:

- For input argument names, `u1`, `u2`, ..., `un`
- For return argument, `y1`

Note This function should be used only for `abs` function replacement. Other functions supported for replacement should use `registerCFunctionEntry`.

Example

In the following example, the `registerCPromotableMacroEntry` function is used to create a function entry for `abs` in a TFL table.

```
hLib = RTW.TflTable;

hLib.registerCPromotableMacroEntry(100, 1, 'abs', 'double', 'abs_prime', ...
                                   'double', '<math_prime.h>', '', '');
```

See Also

`registerCFunctionEntry`

"Alternative Method for Creating Function Entries" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

rtw.codegenObjectives.Objective.removeInheritedCheck

Purpose

Remove inherited checks

Syntax

```
removeInheritedCheck(obj, checkID)
```

Description

`removeInheritedCheck(obj, checkID)` removes an inherited check from the objective definition. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

Input Arguments

obj	Handle to a code generation objective object previously created.
checkID	Unique identifier of the check that you remove from the new objective.

Examples

Remove the **Identify questionable code instrumentation (data I/O)** check from the objective.

```
removeInheritedCheck(obj, 'Identify questionable code instrumentation (data I/O)');
```

See Also

`Simulink.ModelAdvisor`

How To

- "Creating Custom Objectives"
- "About IDs"

rtw.codegenObjectives.Objective.removeInheritedParam

Purpose

Remove inherited parameters

Syntax

```
removeInheritedParam(obj, paramName)
```

Description

`removeInheritedParam(obj, paramName)` removes an inherited parameter from this objective. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another objective includes the parameter, the Code Generation Advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

Input Arguments

obj	Handle to a code generation objective object previously created.
paramName	Parameter that you want to remove from the objective.

Examples

Remove `Inlineparameters` from the objective.

```
removeInheritedParam(obj, 'InlineParams');
```

See Also

`get_param`

How To

- "Creating Custom Objectives"
- "Parameter Command-Line Information Summary"

Purpose

Shut down communications channel with remote processor

Syntax

```
int rtIOStreamClose(
    int streamID
)
```

Arguments

streamID
A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

Description

```
int rtIOStreamClose(
    int streamID
)
```

Call this function to shut down the communications channel and clean up any associated resources.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See Also

`rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, `rtiostream_wrapper`

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_rtiostream`

`rtwdemo_custom_pil`

rtIOStreamOpen

Purpose

Initialize communications channel with remote processor

Syntax

```
int rtIOStreamOpen(
     int    argc,
     void * argv[ ]
)
```

Arguments

argc

Integer argument count, i.e., the number of parameters in `argv[]`

argv[]

An array of pointers to parameters; typically these are null-terminated string parameters, however, this is allowed to be implementation dependent.

Description

```
int rtIOStreamOpen(
     int    argc,
     void * argv[ ]
)
```

This function initializes a communication stream to allow exchange of data between host and target.

The input parameters allows driver-specific parameters to be passed to the communications driver.

If successful, the function returns a nonnegative integer greater than zero, representing a stream handle. A return value of `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See Also

`rtIOStreamSend`, `rtIOStreamRecv`, `rtIOStreamClose`, `rtiostream_wrapper`

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

```
rtwdemo_rtiostream
```

```
rtwdemo_custom_pil
```

Purpose

Receive data from remote processor

Syntax

```
int rtIOStreamRecv(
    int      streamID,
    void   * dst,
   size_t   size,
    size_t * sizeRecvd
)
```

Arguments

streamID
: A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

size
: Size of data to copy into the buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

dst
: A pointer to the start of the buffer where received data must be copied.

sizeRecvd
: The number of units of data received and copied into the buffer `dst` (zero if no data was copied).

Description

```
int rtIOStreamRecv(
    int      streamID,
    void   * dst,
    size_t   size,
    size_t * sizeRecvd
)
```

This function receives data over a communication channel with a remote processor.

A return value of zero indicates success.`RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See also `rtiostreamSend` for implementation and performance considerations.

See Also

`rtIOStreamSend`, `rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStream_wrapper`

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_rtiostream`

`rtwdemo_custom_pil`

rtIOStreamSend

Purpose

Send data to remote processor

Syntax

```
int rtIOStreamSend(
    int          streamID,
   const void * src,
    size_t       size,
   size_t     * sizeSent
)
```

Arguments

streamID
: A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

src
: A pointer to the start of the buffer containing an array of data to transmit

size
: Size of data to transmit. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

sizeSent
: Size of data actually transmitted (always less than or equal to `size`), or zero if no data was transmitted

Description

```
int rtIOStreamSend(
    int          streamID,
    const void * src,
    size_t       size,
    size_t     * sizeSent
)
```

This function sends data over a communication stream with a remote processor.

A return value of zero indicates success.`RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

Implementation and Performance Considerations

The API for `rtIOStream` functions is designed to be independent of the physical layer across which the data is sent. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for the host-target communication.

For a processor-in-the-loop (PIL) application there is no minimum data rate requirement. However, the higher the data rate, the faster the simulation will run.

In general, a communications device driver will require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel may require specification of which available CAN Node should be used.
- A TCP/IP channel may require a port or static IP address to be configured.
- A CAN channel may require the CAN message ID and priority to be specified.

It is the responsibility of the user who implements the `rtIOStream` driver functions to provide this configuration data, for example by hard-coding it, or by supplying arguments to `rtIOStreamOpen`.

See Also

`rtIOStreamOpen`, `rtIOStreamClose`, `rtIOStreamRecv`, `rtiostream_wrapper`

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

```
rtwdemo_rtiostream
```

```
rtwdemo_custom_pil
```

Purpose

Test `rtiostream` shared library methods

Syntax

```
STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')
STATION_ID = rtiostream_wrapper(SHARED_LIB,'open','p1',v1,
   'p2', v2,...)
[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID,
   DATA, SIZE)
[RES,SIZE_RECVD] = rtiostream_wrapper(SHARED_LIB,'recv',ID,
   SIZE)
RES = rtiostream_wrapper(SHARED_LIB,'close',ID)
rtiostream_wrapper(SHARED_LIB, 'unloadlibrary')
```

Description

`rtiostream_wrapper` enables you to access the methods of an `rtiostream` shared library from MATLAB code, for testing purposes.

`STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')` opens an rtIOStream communication channel via a shared library.

If successful, `STATION_ID` is a handle to the channel. A value of `-1` for `STATION_ID` indicates that the attempt to open a channel was unsuccessful. `SHARED_LIB` is the name of a shared library that implements the required `rtIOStream` functions `rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv` and `rtIOStreamClose`. The shared library must be on the system path.

`STATION_ID = rtiostream_wrapper(SHARED_LIB,'open','p1',v1,'p2', v2,...)` opens an rtIOStream communication channel via a shared library, where `p1`, `v1` are additional parameter value pairs. These arguments are implementation dependent, i.e., they are specific to the shared library being called.

`[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID, DATA, SIZE)` sends `DATA` into the communication channel with handle `ID`, and attempts to send `SIZE` bytes. A value of `RES==-1` indicates that an error occurred, and `RES==0` indicates no error. `SIZE_SENT` is the number of bytes accepted by the communication channel. `SIZE_SENT` may be less than `SIZE`, i.e., the requested number of bytes to send.

`[RES,SIZE_RECVD] = rtiostream_wrapper(SHARED_LIB,'recv',ID, SIZE)` receives up to `SIZE` bytes of `DATA` from the communication channel with handle `ID`. A value of `RES==-1` indicates that an error occurred, and `RES==0` indicates no error. `SIZE_RECVD` is the number of bytes actually received from the channel. `SIZE_RECVD` may be less than `SIZE`, i.e., the requested number of bytes to send.

`RES = rtiostream_wrapper(SHARED_LIB,'close',ID)` closes the communication channel with handle `ID`.

`rtiostream_wrapper(SHARED_LIB, 'unloadlibrary')` unloads the `SHARED_LIB`, clearing any persistent data.

See Also

`rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, `rtIOStreamClose`

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_rtiostream`

`rtwdemo_custom_pil`

Purpose

Control and validate AUTOSAR configuration

Description

You can use methods of the `RTW.AutosarInterface` class to configure AUTOSAR code generation and XML import and export options.

Construction

RTW.AutosarInterface	Construct `RTW.AutosarInterface` object

Methods

addIOConf	Add AUTOSAR I/O configuration to model
attachToModel	Attach `RTW.AutosarInterface` object to model
getComponentName	Get XML component name
getDataTypePackageName	Get XML data type package name
getDefaultConf	Get default configuration
getExecutionPeriod	Get runnable execution period
getImplementationName	Get XML implementation name
getInitEventName	Get initial event name
getInitRunnableName	Get initial runnable name
getInterfacePackageName	Get XML interface package name
getInternalBehaviorName	Get XML internal behavior name
getIOAutosarPortName	Get I/O AUTOSAR port name
getIODataAccessMode	Get I/O data access mode
getIODataElement	Get I/O data element name
getIOErrorStatusReceiver	Get receiver port name
getIOInterfaceName	Get I/O interface name

getIOPortNumber	Get I/O AUTOSAR port number
getIOServiceInterface	Get port I/O service interface
getIOServiceName	Get port I/O service name
getIOServiceOperation	Get port I/O service operation
getIsServerOperation	Determine whether server is specified
getPeriodicEventName	Get periodic event name
getPeriodicRunnableName	Get periodic runnable name
getPortDefaultConf	Get port default configuration
getServerInterfaceName	Get name of server interface
getServerOperationPrototype	Get server operation prototype
getServerPortName	Get server port name
getServerType	Determine server type
runValidation	Validate `RTW.AutosarInterface` object against model
setComponentName	Set XML component name
setInitEventName	Set initial event name
setInitRunnableName	Set initial runnable name
setIOAutosarPortName	Set AUTOSAR port name
setIODataAccessMode	Set I/O data access mode
setIODataElement	Set I/O data element
setIOInterfaceName	Set I/O interface name
setIsServerOperation	Indicate that server is specified
setPeriodicEventName	Set periodic event name
setPeriodicRunnableName	Set periodic runnable name
setServerInterfaceName	Set name of server interface

setServerOperationPrototype	Specify operation prototype
setServerPortName	Set server port name
setServerType	Specify server type
syncWithModel	Synchronize configuration with model

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

"Using the Configure AUTOSAR Interface Dialog Box", "Configuring Ports for Basic Software and Error Status Receivers", and "Modifying and Validating an Existing AUTOSAR Interface" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface

Purpose

Construct `RTW.AutosarInterface` object

Syntax

```
autosarInterfaceObject = AutosarInterface()
autosarInterfaceObject = AutosarInterface(model_handle)
autosarInterfaceObject = AutosarInterface(object_name,
   model_handle)
```

Description

`autosarInterfaceObject = AutosarInterface()` creates an `RTW.AutosarInterface` object without specifying a model, and returns a handle to this object.

`autosarInterfaceObject = AutosarInterface(model_handle)` creates an `RTW.AutosarInterface` object with a model specified, and returns a handle to this object. The software sets the name of the `RTW.AutosarInterface` object to `'AutosarInterface'`.

`autosarInterfaceObject = AutosarInterface(object_name, model_handle)` creates an `RTW.AutosarInterface` object with a model specified, and returns a handle to this object. The software sets the name of the `RTW.AutosarInterface` object to *`object_name`*.

Input Arguments

`model_handle`	Handle to Simulink model
`object_name`	Name of `RTW.AutosarInterface` object

Output Arguments

`autosarInterfaceObject`	Handle to newly created `RTW.AutosarInterface` object.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

`RTW.AutosarInterface.attachToModel`

Purpose

Customize code generation objectives

Description

An `rtw.codegenObjectives.Objective` object creates a code generation objective.

Construction

rtw.codegenObjectives.Objective	Create custom code generation objectives

Methods

addCheck	Add checks
addParam	Add parameters
excludeCheck	Exclude checks
modifyInheritedParam	Modify inherited parameter values
register	Register objective
removeInheritedCheck	Remove inherited checks
removeInheritedParam	Remove inherited parameters
setObjectiveName	Specify objective name

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Create a custom objective named `Reduce RAM Example`. The following code is the contents of the `sl_customization.m` file that you create.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();
```

```
end

function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');

%Register the objective
register(obj);

end
```

See Also

`DAStudio.CustomizationManager.ObjectiveCustomizer`

How To

- “Creating Custom Objectives”

Purpose

Create custom code generation objectives

Syntax

```
obj = rtw.codegenObjectives.Objective('objID')
obj = rtw.codegenObjectives.Objective('objID',
'base_objID')
```

Description

`obj = rtw.codegenObjectives.Objective('objID')` creates an objective object, *obj*.

`obj = rtw.codegenObjectives.Objective('objID', 'base_objID')` creates an object, *obj*, for a new objective that is identical to an existing objective. You can then modify the new objective to meet your requirements.

Input Arguments

objID	A permanent, unique identifier for the objective. • You must have *objID*. • The value of *objID* must remain constant. • When you refresh your customizations, if *objID* is not unique, Simulink generates an error.
base_objID	The identifier of the objective that you want to base the new objective on.

Examples

Create a new objective:

```
obj = rtw.codegenObjectives.Objective('ex_ram_1');
```

Create a new objective based on the existing `Execution efficiency` objective:

```
obj = rtw.codegenObjectives.Objective('ex_my_efficiency_1', 'Execution efficiency');
```

How To

- "Creating Custom Objectives"

Purpose

Configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem

Syntax

```
RTW.configSubsystemBuild(block)
```

Description

`RTW.configSubsystemBuild(block)` opens a graphical user interface where you can configure either C function prototype information or C++ encapsulation interface information for right-click builds of a specified nonvirtual subsystem. The appropriate dialog box opens based on the **Language** value selected for your model on the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

To configure and generate C++ encapsulation interfaces for a nonvirtual subsystem, you must

- Select the system target file `ert.tlc` for the model.
- Select the **Language** parameter value `C++ (Encapsulated)` for the model.
- Make sure that the subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

Input Arguments

block	String specifying the name of a nonvirtual subsystem block in an ERT-based Simulink model.

See Also

"Configuring Function Prototypes for Nonvirtual Subsystems" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

"Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Provide parameters to each target connectivity component

Syntax

```
componentArgs = rtw.connectivity.ComponentArgs (componentPath,
   componentCodePath, componentCodeName, applicationCodePath)
```

Description

Syntax of constructor `ComponentArgs`:

```
componentArgs = rtw.connectivity.ComponentArgs
(componentPath, componentCodePath, componentCodeName,
applicationCodePath)
```

You can use the methods of this class to get information about the source component (e.g., the referenced model under test) and the target application (e.g., the PIL application).

For methods, see the following table.

Method	Syntax and Description
getComponentPath	`componentPath = obj.getComponentPath`
	Returns the Simulink system path of the source component (e.g., the path of the referenced model that is under test).
getComponentCodePath	`componentCodePath = obj.getComponentCodePath`
	Returns the Real-Time Workshop Embedded Coder code generation directory path associated with the source component (e.g., the code generation directory of the referenced model that is under test).

Method	Syntax and Description
getComponentCodeName	`componentCodeName = obj.getComponentCodeName`
	Returns the *`modelname`*`.c` name used by Real-Time Workshop Embedded Coder during code generation of the source component.
getApplicationCodePath	`applicationCodePath = obj.getApplicationCodePath`
	Returns the directory path associated with the target application (e.g., the path associated with the PIL application).

See `rtw.connectivity.Config` for more information.

See Also

`rtw.connectivity.Config`

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

Purpose

Define connectivity implementation, comprising builder, launcher, and communicator components

Syntax

```
Config(componentArgs, builder, launcher, communicator)
```

Description

Constructor	Description
ComponentArgs	Wrapper for the connectivity component classes builder, launcher and communicator.

Constructor Arguments	
componentArgs	rtw.connectivity.ComponentArgs object.
builder	rtw.connectivity.Builder (e.g. rtw.connectivity.MakefileBuilder) object.
launcher	rtw.connectivity.Launcher object.
communicator	rtw.connectivity.Communicator (e.g. rtw.connectivity.-RtIOStreamHostCommunicator) object.

Constructor syntax:

```
Config(componentArgs, builder, launcher, communicator)
```

To define a connectivity implementation:

1. You must create a subclass of `rtw.connectivity.Config` that creates instances of your connectivity component classes:
   - rtw.connectivity.MakefileBuilder
   - rtw.connectivity.Launcher

- rtw.connectivity.RtIOStreamHostCommunicator

You can see an example `ConnectivityConfig.m`, used in the demo `rtwdemo_custom_pil`.

2 Define the constructor for your subclass as follows:

```
function this = MyConfig(componentArgs)
```

When Simulink creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you may want to create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

3 After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration, as shown in this example.

```
% call super class constructor to register components
this@rtw.connectivity.Config(componentArgs,...
builder, launcher, communicator);
```

You will register your subclass name (e.g. "`MyPIL.ConnectivityConfig`") to Simulink by using the class `rtw.connectivity.ConfigRegistry`. This uses the `sl_customization.m` mechanism to register your connectivity configuration.

The PIL infrastructure instantiates your subclass as required. The `sl_customization.m` mechanism helps to ensure that a connectivity configuration is suitable for use with a particular PIL component (and its configuration set). It is also possible for the subclass to do extra validation on construction. For example, you can use the `componentPath` returned by the `getComponentPath` method of the

`componentArgs` constructor argument to query and validate parameters associated with the PIL component under test.

For supported hardware implementation settings and other support information, see "SIL and PIL Simulation Support and Limitations" in the Real-Time Workshop Embedded Coder documentation.

See Also

`rtw.connectivity.MakefileBuilder`, `rtw.connectivity.Launcher`, `rtw.connectivity.RtIOStreamHostCommunicator`, `rtw.connectivity.ComponentArgs`

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

rtw.connectivity.ConfigRegistry

Purpose

Register connectivity configuration

Syntax

```
config = rtw.connectivity.ConfigRegistry
config = rtw.connectivity.ConfigRegistry
```

Description

Use this class to register your connectivity configuration with Simulink by using the `sl_customization.m` mechanism. The connectivity configuration is registered by a call to `registerTargetInfo` inside a `sl_customization.m` file.

Create or add to your `sl_customization.m` file as shown in the "Example" on page 3-194 section, and place the file on the MATLAB path. Simulink software reads the `sl_customization.m` when it starts, and registers your connectivity configuration. This step also defines the set of Simulink models that the new connectivity configuration is compatible with.

A connectivity configuration must have a unique name and be associated with a connectivity implementation class (a subclass of `rtw.connectivity.Config`). The properties of the configuration (e.g. `SystemTargetFile`) define the set of Simulink models that the connectivity implementation class is compatible with. The properties are shown in the following table.

Properties of rtw.connectivity.ConfigRegistry

Property Name	Description
ConfigName	Unique string name for this configuration
ConfigClass	Full class name of the connectivity implementation (e.g. `rtw.pil.myConnectivityConfig`) to register.

Properties of rtw.connectivity.ConfigRegistry (Continued)

Property Name	Description
SystemTargetFile	Cell array of strings listing System Target Files that support this `ConfigRegistry`. An empty cell array matches any System Target File. The model's `SystemTargetFile`Configuration Parameter is validated against this cell array to determine if this `ConfigRegistry` is valid for use.
TemplateMakefile	Cell array of strings listing Template Makefiles that support this `ConfigRegistry`. An empty cell array matches any Template Makefile and nonmakefile based targets (`GenerateMakefile`: off). The model's `TemplateMakefile` Configuration Parameter is validated against this cell array to determine if this `ConfigRegistry` is valid for use.
TargetHWDeviceType	Cell array of strings listing Hardware Device Types that support this `ConfigRegistry`. An empty cell array matches any Hardware Device Type. The model's `TargetHWDeviceType`Configuration Parameter is validated against this cell array to determine if this `ConfigRegistry` is valid for use.

Example

The following code shows an example `sl_customization.m` registration. You must use the `sl_customization.m` file structure shown in the example following. You must call the `registerTargetInfo` function exactly as shown.

```
function sl_customization(cm)
% SL_CUSTOMIZATION for PIL connectivity config:...
% mypil.ConnectivityConfig

% Copyright 2008 The MathWorks, Inc.
% $Revision: 1.1.4.9 $

cm.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

config = rtw.connectivity.ConfigRegistry;
config.ConfigName = 'My PIL Example';
config.ConfigClass = 'mypil.ConnectivityConfig';

% match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};
% match the standard ert TMF's
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vc.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};
% match regular 32-bit machines and Custom for e.g. ...
% 64-bit Linux
config.TargetHWDeviceType = {'Generic->32-bit x86 ...
                             compatible'
                            'Generic->Custom'};
```

You must configure the file to perform the following steps when Simulink software starts:

1 Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example,

```
config = rtw.connectivity.ConfigRegistry;
```

2 Assign a connectivity configuration name to the `ConfigName` property of the object. For example,

```
config.ConfigName = 'My PIL Example';
```

3 Associate the connectivity configuration with the connectivity API implementation (created in step 1). For example,

```
config.ConfigClass = 'mypil.ConnectivityConfig';
```

4 Define compatible models for this target connectivity configuration, by setting the `SystemTargetFile`, `TemplateMakefile` and `TargetHWDeviceType` properties of the object. For example,

```
% match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};
% match the standard ert TMF's
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vc.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};
% match regular 32-bit machines and Custom for e.g. ...
% 64-bit Linux
config.TargetHWDeviceType = {'Generic->32-bit x86 ...
                              compatible'
                             'Generic->Custom'};
```

See Also

`rtw.connectivity.Config`

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

```
rtwdemo_custom_pil
```

Purpose

Control downloading, starting and resetting executable on target hardware

Syntax

```
Launcher(componentArgs, builder)
```

Description

Constructor	Syntax
`Launcher`	`Launcher(componentArgs, builder)`

The Launcher component launches an application built by a Builder object. Launcher controls the download, start and reset of the application (e.g. PIL application) associated with a `rtw.connectivity.Builder` object. You must make a subclass and implement the `startApplication` and `stopApplication` methods.

If necessary, you can implement a destructor method that cleans up any resources (e.g., a handle to a 3rd party download tool) when this object is cleared from memory. There is significant flexibility in how the `startApplication` and `stopApplication` methods can be implemented.

See MyPIL.Launcher for an example.

For methods, see the following table.

Method	Syntax and Description
getBuilder	`builder = obj.getBuilder`
	Returns the `rtw.connectivity.Builder` object associated with this Launcher object.

Method	Syntax and Description
startApplication	`obj.startApplication`

	Abstract method that you must implemented in a subclass. Called by Simulink to start execution of the target application, created by the `rtw.connectivity.Builder` object associated with this Launcher object.
	The startApplication method must always reset the application to its initial state by ensuring that external and static (global) variables are zero initialized.
	Use the `getApplicationExecutable` method of the associated `rtw.connectivity.Builder` object to determine the application to start, e.g.,
	`exe = this.getBuilder.get...`
	`ApplicationExecutable`

Method	Syntax and Description
stopApplication	`obj.stopApplication`

	Abstract method that you must implemented in a subclass. Called by Simulink to stop execution of the target application, created by the `rtw.connectivity.Builder` object associated with this Launcher object.
	Use the `getApplicationExecutable` method of the associated `rtw.connectivity.Builder` object to determine the application to stop, e.g.,
	`exe = this.getBuilder.get...`
	`ApplicationExecutable`

getComponentArgs	`componentArgs = obj.getComponentArgs`
	Returns the `rtw.connectivity.ComponentArgs` object associated with this Launcher object.

See Also

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

rtw.connectivity.MakefileBuilder

Purpose

Configure makefile-based build process

Syntax

```
MakefileBuilder(componentArgs, targetApplicationFramework,
   exeExtension)
```

Description

Constructor	Description
MakefileBuilder	Control makefile-based build process.

Constructor Arguments	
componentArgs	rtw.connectivity.ComponentArgs
TargetApplicationFramework	rtw.pil.RtIOStream-ApplicationFramework (e.g. MyPIL.TargetFramework)
exeExtension	Filename extension of an executable for the target system. The extension depends on the makefile and compiler that are called by the `MakefileBuilder`. These are defined by the template makefile specified by the source component (e.g., the referenced model under test). For an embedded target the extension may be `'.elf'`, `'.abs'`, `'.sre'`, `'.hex'`, or others. For a Windows® host-based target the extension is `'.exe'`. For a UNIX® host-based target the extension is empty, `''`.

Constructor syntax:

```
MakefileBuilder(componentArgs, targetApplicationFramework,
exeExtension)
```

MakefileBuilder controls the customizable makefile-based build process supporting the creation of custom applications (e.g. a PIL application) that interface with a Simulink component such as a referenced model (represented as a collection of binary libraries).

To build the PIL application, you must provide a template makefile that includes the target MAKEFILEBUILDER_TGT. You can use any of the standard TMF files, e.g., `ert_unix.tmf` or `ert_vc.tmf`.

See Also

`rtw.pil.RtIOStreamApplicationFramework`, `rtw.connectivity.ComponentArgs`

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

rtw.connectivity.RtIOStreamHostCommunicator

Purpose

Configure host-side communications

Syntax

```
RtIOStreamHostCommunicator (componentArgs, launcher,
   rtiostreamLib)
```

Description

Constructor	Description
RtIOStreamHostCommunicator	Configure host-side communications with the target by loading and initializing a shared library that implements the `rtiostream` functions.

Constructor Arguments	
componentArgs	A `rtw.connectivity.ComponentArgs` object.
launcher	A `rtw.connectivity.Launcher` object.
rtiostreamLib	An `rtiostream` shared library that implements the host side of host-target communications.

Constructor syntax:

```
RtIOStreamHostCommunicator (componentArgs, launcher,
rtiostreamLib)
```

This class configures host-side communications with the target by loading and initializing a shared library that implements the `rtiostream` functions.

Real-Time Workshop Embedded Coder provides an implementation of this shared library to support TCP/IP communications between host and target (all platforms), as well as a Windows only version for serial

communications. With TCP/IP or serial, you need only supply the target-side drivers.

For other communications protocols (e.g. USB), you must supply an appropriate shared library for the host-side of the communications link as well as the target-side drivers.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have two options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments to supply to the `rtiostream` shared library. This is sufficient in most cases.
- Alternatively, create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. This may be necessary when more complex configuration is required. For example, the demo subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the TCP/IP port number on which the executable application is serving, or you could use a subclass to specify a serial port number, or specify verbose or silent operation.

Methods	
setTimeoutRecvSecs	Sets the timeout value for reading data.
`hostCommunicator.setTimeoutRecvSecs(`*`timeout`*`)` configures data reading to time out if no new data is received for a period of greater than `timeout` seconds.	

Methods	
setInitCommsTimeout	Sets the timeout value for initial setup of the communications channel.

`hostCommunicator.setInitCommsTimeout(`*`timeout`*`)` For some targets you may need to set a timeout value for initial setup of the communications channel. For example, the target processor may take a few seconds before it is ready to open its side of the communications channel. If you set a nonzero timeout value then the communicator repeatedly tries to open the communications channel until the timeout value is reached.

See Also

`rtw.connectivity.ComponentArgs`, `rtw.connectivity.Launcher`, `rtiostream_wrapper`

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

Purpose

Get handle to model-specific C++ encapsulation interface control object

Syntax

```
obj = RTW.getEncapsulationInterfaceSpecification(modelName)
```

Description

`obj = RTW.getEncapsulationInterfaceSpecification(modelName)` returns a handle to a model-specific C++ encapsulation interface control object.

Input Arguments

modelName	String specifying the name of a loaded ERT-based Simulink model.

Output Arguments

obj	Handle to the C++ encapsulation interface control object associated with the specified model. If the model does not have any associated C++ encapsulation interface control object, the function returns `[]`.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ encapsulation interface modifications. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get handle to model-specific C prototype function control object

Syntax

```
obj = RTW.getFunctionSpecification(modelName)
```

Description

`obj = RTW.getFunctionSpecification(modelName)` returns a handle to the model-specific C function prototype control object.

Input Arguments

modelName	String specifying the name of a loaded ERT-based Simulink model.

Output Arguments

obj	Handle to the model-specific C prototype function control object associated with the specified model. If the model does not have any associated function control object, the function returns `[]`.

Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. Once you validate and apply your changes, you can generate code based on your C function prototype modifications. See "Configuring Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.ModelCPPArgsClass class

Superclasses

ModelCPPClass

Purpose

Control C++ encapsulation interfaces for models using I/O arguments style step method

Description

The `ModelCPPArgsClass` class provides objects that describe C++ encapsulation interfaces for models using an I/O arguments style step method. Use the `attachToModel` method to attach a C++ encapsulation interface to a loaded ERT-based Simulink model.

Construction

RTW.ModelCPPArgsClass	Create C++ encapsulation interface object for configuring model class with I/O arguments style step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following methods.

getArgCategory	Get argument category for Simulink model port from model-specific C++ encapsulation interface
getArgName	Get argument name for Simulink model port from model-specific C++ encapsulation interface
getArgPosition	Get argument position for Simulink model port from model-specific C++ encapsulation interface
getArgQualifier	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface

runValidation	Validate model-specific C++ encapsulation interface against Simulink model
setArgCategory	Set argument category for Simulink model port in model-specific C++ encapsulation interface
setArgName	Set argument name for Simulink model port in model-specific C++ encapsulation interface
setArgPosition	Set argument position for Simulink model port in model-specific C++ encapsulation interface
setArgQualifier	Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ encapsulation interface modifications. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create C++ encapsulation interface object for configuring model class with I/O arguments style step method

Syntax

```
obj = RTW.ModelCPPArgsClass
```

Description

`obj = RTW.ModelCPPArgsClass` returns a handle, *obj*, to a newly created object of class `RTW.ModelCPPArgsClass`.

Output Arguments

obj	Handle to a newly created C++ encapsulation interface object for configuring a model class with an I/O arguments style step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelCPPClass class

Purpose

Control C++ encapsulation interfaces for models

Description

The `ModelCPPClass` class is the base class for the classes `RTW.ModelCPPArgsClass` and `RTW.ModelCPPVoidClass`, which provide objects that describe C++ encapsulation interfaces for models using either an I/O arguments style step method or a `void-void` style step method. Use the `attachToModel` method to attach a C++ encapsulation interface to a loaded ERT-based Simulink model.

Construction

To access the methods of this class, use the constructor for either `RTW.ModelCPPArgsClass` or `RTW.ModelCPPVoidClass`.

Methods

attachToModel	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
getClassName	Get class name from model-specific C++ encapsulation interface
getDefaultConf	Get default configuration information for model-specific C++ encapsulation interface from Simulink model
getNumArgs	Get number of step method arguments from model-specific C++ encapsulation interface
getStepMethodName	Get step method name from model-specific C++ encapsulation interface

setClassName	Set class name in model-specific C++ encapsulation interface
setStepMethodName	Set step method name in model-specific C++ encapsulation interface

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ encapsulation interface modifications. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelCPPVoidClass class

Superclasses

ModelCPPClass

Purpose

Control C++ encapsulation interfaces for models using `void-void` style step method

Description

The `ModelCPPVoidClass` class provides objects that describe C++ encapsulation interfaces for models using a `void-void` style step method. Use the `attachToModel` method to attach a C++ encapsulation interface to a loaded ERT-based Simulink model.

Construction

RTW.ModelCPPVoidClass	Create C++ encapsulation interface object for configuring model class with `void-void` style step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following method.

runValidation	Validate model-specific C++ encapsulation interface against Simulink model

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ encapsulation interface modifications. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelCPPVoidClass

Purpose

Create C++ encapsulation interface object for configuring model class with `void-void` style step method

Syntax

```
obj = RTW.ModelCPPVoidClass
```

Description

`obj = RTW.ModelCPPVoidClass` returns a handle, *`obj`*, to a newly created object of class `RTW.ModelCPPVoidClass`.

Output Arguments

`obj`	Handle to a newly created C++ encapsulation interface object for configuring a model class with a `void-void` style step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Describe signatures of functions for model

Description

A `ModelSpecificCPrototype` object describes the signatures of the step and initialization functions for a model. You must use this in conjunction with the `attachToModel` method.

Construction

RTW.ModelSpecificCPrototype	Create model-specific C prototype object

Methods

addArgConf	Add argument configuration information for Simulink model port to model-specific C function prototype
attachToModel	Attach model-specific C function prototype to loaded ERT-based Simulink model
getArgCategory	Get argument category for Simulink model port from model-specific C function prototype
getArgName	Get argument name for Simulink model port from model-specific C function prototype
getArgPosition	Get argument position for Simulink model port from model-specific C function prototype
getArgQualifier	Get argument type qualifier for Simulink model port from model-specific C function prototype

getDefaultConf	Get default configuration information for model-specific C function prototype from Simulink model
getFunctionName	Get function name from model-specific C function prototype
getNumArgs	Get number of function arguments from model-specific C function prototype
getPreview	Get model-specific C function prototype code preview
runValidation	Validate model-specific C function prototype against Simulink model
setArgCategory	Set argument category for Simulink model port in model-specific C function prototype
setArgName	Set argument name for Simulink model port in model-specific C function prototype
setArgPosition	Set argument position for Simulink model port in model-specific C function prototype
setArgQualifier	Set argument type qualifier for Simulink model port in model-specific C function prototype
setFunctionName	Set function name in model-specific C function prototype

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

The code below creates a function control object, `a`, and uses it to add argument configuration information to the model.

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a function control object
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the function control object to the model
attachToModel(a,gcs)
```

Alternatives

You can create a function control object using the Model Interface dialog box.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.ModelSpecificCPrototype

Purpose

Create model-specific C prototype object

Syntax

```
obj = RTW.ModelSpecificCPrototype
```

Description

`obj = RTW.ModelSpecificCPrototype` creates a handle, *obj*, to an object of class `RTW.ModelSpecificCPrototype`.

Output Arguments

obj	Handle to model specific C prototype object.

Examples

Create a function control object, `a`, and use it to add argument configuration information to the model:

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a function control object
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the function control object to the model
attachToModel(a,gcs)
```

Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. See "Configuring Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

rtw.pil.RtIOStreamApplicationFramework

Purpose

Configure target-side communications

Syntax

```
applicationFramework = rtw.pil.RtIOStreamApplicationFramework(
   componentArgs)
```

Description

Constructor	Description
RtIOStreamApplicationFramework	Specify target-specific libraries and source files that are required to build the executable.

Constructor Argument	
componentArgs	A `rtw.connectivity.ComponentArgs` object.

Constructor syntax:

```
applicationFramework =
rtw.pil.RtIOStreamApplicationFramework(componentArgs)
```

You must create a subclass of `rtw.pil.RtIOStreamApplicationFramework`. The purpose of this class is to specify target-specific libraries and source files that are required to build the executable for the PIL application. These libraries and source files must include the device drivers that implement the target-side of the `rtiostream` communications channel. See also `rtiostream_wrapper`.

The class provides an `RTW.BuildInfo` object containing PIL-specific files (including a PIL main) that will be combined with the PIL component libraries, by the `rtw.connectivity.MakefileBuilder`, to create the PIL application. You must make a subclass and add source files, libraries, include paths and preprocessor defines that are required

to implement the `rtiostream` target communications interface to the `RTW.BuildInfo` object (access via `getBuildInfo` method).

For methods, see the following table.

Method	Syntax and Description
`getComponentArgs`	`componentArgs = obj.getComponentArgs`
	Returns the `rtw.connectivity.ComponentArgs` object associated with this object.
`getBuildInfo`	`buildInfo = obj.getBuildInfo`
	Returns the `RTW.BuildInfo` object associated with this object.
`addPILMain`	`obj.addPILMain(type)`
	To build the PIL application you must specify a `main.c` file. Use the `addPILMain` method to add one of the two provided files to the application framework. Use the `type` argument to specify `'target'` or `'host'`, depending on which one of the following example PIL `main.c` files you want to use. 1) To specify a `main.c` adapted for on-target PIL and suitable for most PIL implementations, enter: `obj.addPILMain(`target')` 2) To specify a `main.c` adapted for host-based PIL, for example, as

Method	Syntax and Description
	used in the `mypil` host example, enter: `obj.addPILMain(`host')`

See Also

`rtw.connectivity.ComponentArgs`, `rtiostream_wrapper`

"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for a Target" in the Real-Time Workshop Embedded Coder documentation.

"Build Information Object" in the Real-Time Workshop documentation.

`rtwdemo_custom_pil`

Purpose

Execute CGV object

Syntax

```
result = cgvObj.run()
```

Description

result = *cgvObj*.run() executes the model once for each input data that you added to the object. *result* is a boolean value that indicates whether the run completed without execution error. *cgvObj* is a handle to a cgv.CGV object.

After each execution of the model, the object captures and writes the following metadata to a file in the output folder:

ErrorDetails — If errors occur, the error information.
status — The execution status.
ver — Version information for MathWorks products.
hostname — Name of computer.
dateTime — Date and time of execution.
warnings — If warnings occur, the warning messages.
username — Name of user.
runtime — The amount of time that lapsed for the execution.

Note:

- Only call run once for each cgv.CGV object. A
- The cgv.CGV methods that set up the object are ignored after a call to run. See the cgv.CGV class for details.
- You can call run once without first calling cgv.CGV.addInputData. However, it is recommended that you save all of the required data for execution to a MAT-file, including the model inputs and parameters. Then use cgv.CGV.addInputData to pass the MAT-file to the CGV object before calling run.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

RTW.AutosarInterface.runValidation

Purpose

Validate `RTW.AutosarInterface` object against model

Syntax

```
[Status, Message] = autosarInterfaceObj.runValidation
```

Description

`[Status, Message] = autosarInterfaceObj.runValidation` runs a validation check for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object. This check is made against the model to which *`autosarInterfaceObj`* is attached.

Before calling `runValidation`, you must call `attachToModel`.

The method `runValidation` performs the checks described in the following tables. The first table describes validation checks for all AUTOSAR use cases, and the second table describes specific validation checks when exporting multiple runnable entities.

Validation Checks

Group	Check
Valid names and paths	Runnable names and event names must all be unique, and must be valid AUTOSAR short name identifiers (see definition 1 following).
	AUTOSAR port, interface, and data element names must be valid AUTOSAR short name identifiers (see definition 1 following).
	AUTOSAR XML options for the component name, internal behavior name, and implementation name must be valid AUTOSAR path and short name identifiers (see definition 2 following).
	AUTOSAR XML options for the interface package name and data type package name must be valid AUTOSAR path identifiers (see definition 3 following).

Validation Checks (Continued)

Group	Check
Valid names and paths for sender/receiver ports	For sender/receiver ports (Implicit or explicit data access mode): • Simulink ports may have duplicated `AUTOSAR port` names, however the AUTOSAR `Interface` name must also be the same. • A Simulink inport and an outport cannot have the same `AUTOSAR port` name. • For any duplicated `AUTOSAR port` name and AUTOSAR `Interface` name, the `Data element` names must be unique. • Sender/receiver ports `AUTOSAR port` name cannot be the same as the `ServiceName` of a basic software port. • Sender/receiver ports `AUTOSAR port` name and `Interface` cannot be the same as the port name or interface of a calibration object. • Sender/receiver ports `Interface` plus XML Option `Interface package` (e.g., of the form `AUTOSAR/Service/servicename`) cannot be the same as the `ServiceInterface` of a basic software port.

Validation Checks (Continued)

Group	Check
Valid names and paths for basic software ports	For basic software ports: • `ServiceName` and `ServiceOperation` must be valid AUTOSAR short name identifiers (see definition 1 following); and `ServiceInterface` must be a valid AUTOSAR path identifier (see definition 3 following). • Simulink ports may have duplicated `ServiceName`, however the `ServiceInterface` must also be the same. • For any duplicated `ServiceName` and `ServiceInterface`, the `ServiceOperation` must be unique. • For duplicated `ServiceOperation` and `ServiceInterface`, the `ServiceName` must be unique. • Basic software port `ServiceName` name and `ServiceInterface` cannot be the same as the port name or interface of a calibration object.
Unsupported features	Model must not contain custom code blocks.
	Model must not contain continuous time.
	Model must not contain noninlined S-functions.
	Model must not contain nonfinite numbers.
	Model must not contain complex numbers.
	Model must not contain multitasking
	Model must not contain asynchronous rates
	Storage class of root I/O ports must be `auto`.
	I/O must be 1D or scalar.

Validation Checks (Continued)

Group	Check
	The sample time of a runnable must be a positive real scalar. Sample times with offset, e.g. [2 1], cause an error message.
Error status validation	An error status inport cannot point to itself (i.e., cannot specify itself as the inport for which it permits access to error status).
	Error status inports can only be defined to correspond to other inports that have Data Access Mode set to ImplicitReceive or ExplicitReceive
	Each receiver port can have only one error status port designate it as its error status.

Multiple Runnable Validation Checks

Group	Check
Wrapper subsystem validation when exporting multiple runnables. The "wrapper subsystem" is the top diagram runnables are exported from.	“Top-level” function-call subsystems (that are in the top diagram of the wrapper subsystem) must not be reusable functions. Their `'RTW System code'` option must be set to `'Auto'`, `'Function'` or `'Inline'`.
	Top-level function-call subsystems cannot emit function calls.
	The only subsystems allowed at the top diagram are function-call subsystems, and empty subsystems (e.g., subsystems that contain no executable blocks, which may be used to display text in the model, or to double-click for help callback.)

Multiple Runnable Validation Checks (Continued)

Group	Check
	Top-level function-call subsystems cannot have wide trigger ports.
	A signal connected to an outport of the wrapper subsystem cannot have multiple destinations. The signal must have one destination that is uniquely a sender, service, or interrunnable variable.
	A signal connected to an outport of the wrapper subsystem cannot have an inport of that subsystem as its source.
	All data store memory blocks referenced from subsystems must be contained in the subsystems, to prevent data integrity issues.
	All lines must be contiguous. No line in the wrapper subsystem can be an output of a virtual Bus Creator or Mux block
	Constant blocks are not allowed in the wrapper subsystem.
	No Mux, or Demux blocks are allowed in the wrapper subsystem, because the signals being passed via the runnable I/O must be contiguous and have an address at the base of the array.

Multiple Runnable Validation Checks (Continued)

Group	Check
Wrapper level Merge block validation	Merge blocks have some restrictions at wrapper level: • A merge block is only allowed in the wrapper subsystem when the merge block output is connected to a diagram outport (not another Merge block). • The input to a Merge block in the wrapper subsystem must be connected to a function-call subsystem outport. • The input to a Merge block in the wrapper subsystem does not need a label. • A merge block in the wrapper subsystem cannot merge signals of unequal widths. • You cannot connect a Merge block in the wrapper subsystem to more than one outport of any given function-call subsystem.
Other multiple runnable validation checks	All runnable names, event names, and interrunnable variable names must be unique. Lines representing interrunnable variables must be labelled with valid AUTOSAR short name identifiers. No goto-from pairs are allowed because then the signal label is not unique.
	Interrunnable variables cannot be structs. All interrunnable variables must be scalar, noncomplex types. This is required by the AUTOSAR specification. Signal lines that connect two top-level function-call subsystems represent interrunnable variables.

Multiple Runnable Validation Checks (Continued)

Group	Check
	Function-call subsystem output cannot be connected to its own input. An output of a function-call subsystem inside the wrapper subsystem cannot be connected to an input of same subsystem.
	The blocks in the top diagram of the wrapper subsystem must not have unconnected ports.
	Any top-level input that is Explicit Receive, Error Status, or Basic Software Service cannot be connected to more than one inport of any given function-call subsystem.
	The sample time of the inport associated with an error status must be the same sample time as its corresponding data port.
	Each function call subsystem being exported as a runnable entity must specify an AUTOSAR interface.

Output Arguments

`Status`	Status flag indicating whether the configuration is valid. If valid, `Status` is true; otherwise, it is false.
`Message`	If `Status` is false, `Message` explains why the configuration is invalid.

Definitions

The following are requirements for identifiers:

1 *AUTOSAR short name identifiers* must be composed of at most 32 characters, must begin with a letter, and can contain only

letters, numbers, and underscore characters. For example, `this_is_valid123`.

2. *AUTOSAR path and short name identifiers* must contain at least two path delimiter "/" characters, e.g., `/path/shortname`. Strings in between the path delimiters must be composed of at most 32 characters, must begin with a letter, and can contain only letters, numbers, and underscore characters.

3. *AUTOSAR path identifiers* must contain at least one path delimiter "/" characters, e.g., `/path`. Strings in between the path delimiters must be composed of at most 32 characters, must begin with a letter and can contain only letters, numbers, and underscore characters.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelCPPArgsClass.runValidation

Purpose

Validate model-specific C++ encapsulation interface against Simulink model

Syntax

```
[status, msg] = runValidation(obj)
```

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ encapsulation interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getEncapsulationInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.
msg	If *status* is false, *msg* contains a string of information describing why the configuration is invalid.

Alternatives

To validate a C++ encapsulation interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step

function configuration. The **Validation** pane displays success or failure status and an explanation of any failure. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelCPPVoidClass.runValidation

Purpose

Validate model-specific C++ encapsulation interface against Simulink model

Syntax

```
[status, msg] = runValidation(obj)
```

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ encapsulation interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getEncapsulationInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPVoidClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.
msg	If *status* is false, *msg* contains a string of information describing why the configuration is invalid.

Alternatives

To validate a C++ encapsulation interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step

function configuration. The **Validation** pane displays success or failure status and an explanation of any failure. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.runValidation

Purpose

Validate model-specific C function prototype against Simulink model

Syntax

```
[status, msg] = runValidation(obj)
```

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getFunctionSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.

Output Arguments

status	True for a valid configuration; false otherwise.
msg	If *status* is false, *msg* contains a string explaining why the configuration is invalid.

Alternatives

Click the **Validate** button in the Model Interface dialog box to run a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Set argument category for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgCategory(obj, portName, category)
```

Description

`setArgCategory(obj, portName, category)` sets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.
portName	String specifying the unqualified name of an inport or outport in your Simulink model.
category	String specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — to be set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, the change causes the argument to move to the first argument position when `attachToModel` or `runValidation` is called.

Alternatives

To set argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where

you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories that you can examine and modify. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument category for Simulink model port in model-specific C function prototype

Syntax

```
setArgCategory(obj, portName, category)
```

Description

`setArgCategory(obj, portName, category)` sets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype` or *obj* = `RTW.getFunctionSpecification(`*modelName*`)`.
portName	String specifying the unqualified name of an inport or outport in your Simulink model.
category	String specifying the argument category, `'Value'` or `'Pointer'`, that you set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, it causes the argument to move to the first argument position when you call `RTW.ModelSpecificCPrototype.attachToModel` or `RTW.ModelSpecificCPrototype.runValidation`.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument categories. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Set argument name for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgName(obj, portName, argName)
```

Description

`setArgName(obj, portName, argName)` sets the argument name that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.
portName	String specifying the name of an inport or outport in your Simulink model.
argName	String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

To set argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names that you can examine and modify. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument name for Simulink model port in model-specific C function prototype

Syntax

```
setArgName(obj, portName, argName)
```

Description

`setArgName(obj, portName, argName)` sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
argName	String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument names. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.ModelCPPArgsClass.setArgPosition

Purpose

Set argument position for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgPosition(obj, portName, position)
```

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface. The specified argument is then moved to the specified position, and other arguments shifted by one position accordingly.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
position	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

To set argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the

Configure C++ Encapsulation Interface button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions that you can examine and modify. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.ModelSpecificCPrototype.setArgPosition

Purpose

Set argument position for Simulink model port in model-specific C function prototype

Syntax

```
setArgPosition(obj, portName, position)
```

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype. The specified argument moves to the specified position, and other arguments shift by one position accordingly.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
position	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument position. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

Purpose

Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgQualifier(obj, portName, qualifier)
```

Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.
portName	String specifying the name of an inport or outport in your Simulink model.
qualifier	String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — to be set for the specified Simulink model port.

Alternatives

To set argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers that you can examine and modify. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument type qualifier for Simulink model port in model-specific C function prototype

Syntax

```
setArgQualifier(obj, portName, qualifier)
```

Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
qualifier	String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — to be set for the specified Simulink model port.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument qualifiers. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.ModelCPPClass.setClassName

Purpose

Set class name in model-specific C++ encapsulation interface

Syntax

```
setClassName(obj, clsName)
```

Description

`setClassName(obj, clsName)` sets the class name in the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass`, *obj* = `RTW.ModelCPPVoidClass`, or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.
clsName	String specifying a new name for the class described by the specified model-specific C++ encapsulation interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the model class name, which you can examine and modify. In the `void-void step method` view, you can examine and modify the model class name without having to click a button. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.setComponentName

Purpose

Set XML component name

Syntax

```
autosarInterfaceObj.setComponentName(componentName)
```

Description

`autosarInterfaceObj.setComponentName(componentName)` sets the XML component name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`componentName`	XML component name for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.getComponentName`

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set XML file dependencies

Syntax

```
importerObj.setDependencies(dependencies)
```

Description

`importerObj.setDependencies(dependencies)` sets the XML file dependencies associated with the `arxml.importer` object, *`importerObj`*.

Input Arguments

`dependencies`

Can be:

- a cell array of strings (for a list of dependencies)
- a char array (for a single dependency)
- or the empty array [] (for removing any dependency)

Note All atomic software components described in the XML file dependencies are ignored.

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

arxml.importer.setFile

Purpose

Set XML file name for `arxml.importer` object

Syntax

`importerObj``.setFile(`*`filename`*`)`

Description

`importerObj``.setFile(`*`filename`*`)` sets the name of the XML file associated with the `arxml.importer` object, `importerObj`.

Input Arguments

`filename`	XML file name. Only atomic software components described in this file can be imported.

See Also

"Importing an AUTOSAR Software Component" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set function name in model-specific C function prototype

Syntax

```
setFunctionName(obj, fcnName, fcnType)
```

Description

`setFunctionName(obj, fcnName, fcnType)` sets the step or initialization function name in the specified function control object.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype` or *obj* = `RTW.getFunctionSpecification(`*modelName*`)`.
fcnName	String specifying a new name for the function described by the function control object. The argument must be a valid C identifier.
fcnType	Optional. String specifying which function to name. Valid strings are `'step'` and `'init'`. If *fcnType* is not specified, sets the step function name.

Alternatives

Use the **Initialize function name** and **Step function name** fields in the Model Interface dialog box to specify function names. See "Model Specific C Prototypes View" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Controlling Generation of Function Prototypes" — Explains how to configure model function prototypes in generated code

RTW.AutosarInterface.setInitEventName

Purpose

Set initial event name

Syntax

```
autosarInterfaceObj.setInitEventName(initEventName)
```

Description

`autosarInterfaceObj.setInitEventName(initEventName)` sets the initial event name for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`initEventName`	Initial event name for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.getInitEventName`

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set initial runnable name

Syntax

```
autosarInterfaceObj.setInitRunnableName(initRunnableName)
```

Description

`autosarInterfaceObj.setInitRunnableName(initRunnableName)` sets the initial runnable name for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`initRunnableName`	Initial runnable name for *`autosarInterfaceObj`*.

See Also

`RTW.AutosarInterface.getInitRunnableName`

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.setIOAutosarPortName

Purpose

Set AUTOSAR port name

Syntax

```
autosarInterfaceObj.setIOAutosarPortName(portName,
   autosarPort)
```

Description

autosarInterfaceObj`.setIOAutosarPortName(`*portName*`,`*autosarPort*`)` updates the AUTOSAR port name in the configuration for the specified port.

autosarInterfaceObj is a model-specific `RTW.AutosarInterface` object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of inport/outport (string)
autosarPort	AUTOSAR port name for *portName* (string).

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set I/O data access mode

Syntax

```
autosarInterfaceObj.setIODataAccessMode(portName,
   dataAccessMode)
```

Description

`autosarInterfaceObj.setIODataAccessMode(portName,dataAccessMode)` sets the data access mode in the configuration for the specified port.

autosarInterfaceObj is a model-specific `RTW.AutosarInterface` object.

Input Arguments

portName	Name of inport/outport (string).
dataAccessMode	Data access mode (string). Can be one of the following: • `'ImplicitSend'` • `'ImplicitReceive'` • `'ExplicitSend'` • `'ExplicitReceive'`

See Also

`RTW.AutosarInterface.getIODataAccessMode`

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.setIODataElement

Purpose

Set I/O data element

Syntax

```
autosarInterfaceObj.setIODataElement(portName,dataElement)
```

Description

`autosarInterfaceObj.setIODataElement(portName,dataElement)` updates the name of the I/O data element in the configuration for the specified port.

autosarInterfaceObj is a model-specific `RTW.AutosarInterface` object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of the inport/outport (string).
dataElement	Name of the I/O data element for *portName* (string).

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set I/O interface name

Syntax

```
autosarInterfaceObj.setIOInterfaceName(portName,
   interfaceName)
```

Description

autosarInterfaceObj.setIOInterfaceName(*portName*,*interfaceName*) updates the I/O interface name in the configuration for the specified port.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of inport/outport (string).
interfaceName	Name of I/O interface for *portName* (string).

See Also

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.setIsServerOperation

Purpose

Indicate that server is specified

Syntax

```
autosarInterfaceObj.setIsServerOperation(isServerOperation)
```

Description

`autosarInterfaceObj.setIsServerOperation(isServerOperation)` sets the value of the property `'isServerOperation'` in *`autosarInterfaceObj`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`isServerOperation`	True or false (default). If true, indicates that a server is specified in *`autosarInterfaceObj`*.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set name space for C++ function entry in TFL table

Syntax

```
setNameSpace(hEntry, nameSpace)
```

Arguments

hEntry

Handle to a TFL function entry previously returned by one of the following:

- *hEntry* = `RTW.TflCFunctionEntry`
- *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from `RTW.TflCFunctionEntry`
- A call to the `registerCPPFunctionEntry` function

nameSpace

String specifying the name space in which the implementation function for the C++ function entry is defined.

Description

The `setNameSpace` function specifies the name space for a C++ function entry in a TFL table. During code generation, if the TFL function entry is matched, the software emits the name space in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`).

If you created the function entry using *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = *MyCustomFunctionEntry* (that is, not using `registerCPPFunctionEntry`), then, before calling the `setNameSpace` function, you must enable C++ support for the function entry by calling the `enableCPP` function.

Example

In the following example, the `setNameSpace` function is used to set the name space for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                  'Key',                    'sin', ...
                                  'Priority',               100, ...
                                  'ImplementationName',     'sin', ...
```

```
                                    'ImplementationHeaderFile', 'cmath' );
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

See Also

`enableCPP`, `registerCPPFunctionEntry`

"Example: Mapping Math Functions to Target-Specific Implementations" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Specify objective name

Syntax

`setObjectiveName(obj, objName)`

Description

`setObjectiveName(obj, objName)` specifies a name for the objective. The Configuration Set Objectives dialog box displays the name of the objective.

Input Arguments

obj	Handle to a code generation objective object previously created.
objName	Optional string that indicates the name of the objective. If you do not specify an objective name, the Configuration Set Objectives dialog box displays the objective ID for the objective name.

Examples

Name the objective Reduce RAM Example:

```
setObjectiveName(obj, 'Reduce RAM Example');
```

How To

- "Creating Custom Objectives"

cgv.CGV.setOutputDir

Purpose

Specify folder

Syntax

```
cgvObj.setOutputDir('path')
cgvObj.setOutputDir('path', 'overwrite', 'on')
```

Description

`cgvObj.setOutputDir('path')` is an optional method that specifies a location where the object writes all output and metadata files for execution. *`cgvObj`* is a handle to a `cgv.CGV` object. *`path`* is the absolute or relative path to the folder. If the path does not exist, the object attempts to create the folder. If you do not call `setOutputDir`, the object uses the current working folder.

`cgvObj.setOutputDir('path', 'overwrite', 'on')` includes the property and value pair to allow read-only files in the working directory to be overwritten. The default value for `'overwrite'` is *`'off'`*.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

Purpose

Specify output data file name

Syntax

`cgvObj.setOutputFile(InputIndex,OutputFile)`

Description

`cgvObj.setOutputFile(InputIndex,OutputFile)` is an optional method that changes the default file name for the output data. *cgvObj* is a handle to a `cgv.CGV` object. *InputIndex* is a unique numeric identifier that specifies which output data to write to the file. The *InputIndex* is associated with specific input data.*OutputFile* is the name of the file, with or without the `.mat` extension.

How To

- "Verifying Numerical Equivalence of Results with Code Generation Verification API"

RTW.AutosarInterface.setPeriodicEventName

Purpose

Set periodic event name

Syntax

```
autosarInterfaceObj.setPeriodicEventName(periodicEventName)
```

Description

`autosarInterfaceObj.setPeriodicEventName(periodicEventName)` sets the name of the periodic event for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`periodicEventName`	Name of the periodic event for *`autosarInterfaceObj`*.

See Also

`RTW.AutosarInterface.getPeriodicEventName`

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set periodic runnable name

Syntax

```
autosarInterfaceObj.setPeriodicRunnableName(periodicRunnableName)
```

Description

`autosarInterfaceObj.setPeriodicRunnableName(periodicRunnableName)` sets the name of the periodic runnable for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`periodicRunnableName`	Name of periodic runnable for *`autosarInterfaceObj`*.

See Also

`RTW.AutosarInterface.getPeriodicRunnableName`

"Using the Configure AUTOSAR Interface Dialog Box" in the Real-Time Workshop Embedded Coder documentation

setReservedIdentifiers

Purpose

Register specified reserved identifiers to be associated with TFL table

Syntax

```
setReservedIdentifiers(hTable, ids)
```

Arguments

hTable

Handle to a TFL table previously returned by *hTable* = `RTW.TflTable`.

ids

Structure specifying reserved keywords to be registered in the TFL table. The structure must contain the following:

- `LibraryName` element, a string that specifies a TFL name: `'ANSI'`, `'ISO'`,`'GNU'`, or a TFL name of your choice.
- `HeaderInfos` element, a structure or cell array of structures containing
  - `HeaderName` element, a string that specifies the header file in which the identifiers are declared
  - `ReservedIds` element, a cell array of strings that specifies the names of the identifiers to be registered as reserved keywords

For example,

```
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

Description

In a TFL table, each function implementation name defined by a table entry will be registered as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

The `setReservedIdentifiers` function allows you to register up to four reserved identifier structures in a TFL table. One set of reserved identifiers can be associated with an arbitrary TFL, while the other

three (if present) must be associated with ANSI®[1], ISO®[2], or GNU®[3] libraries.

For information about generating a list of reserved identifiers for the TFL that you are using to generate code, see "Real-Time Workshop Target Function Library Keywords" in the Real-Time Workshop documentation.

Example

In the following example, `setReservedIdentifiers` is used to register four reserved identifier structures, for `'ANSI'`, `'ISO'`,`'GNU'`, and `'My Custom TFL'`, respectively.

```
hLib = RTW.TflTable;

% Create and register TFL entries here

.
.
.

% Create and register reserved identifiers
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{1}.HeaderInfos{2}.HeaderName = 'foo.h';
d{1}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{2}.LibraryName = 'ISO';
d{2}.HeaderInfos{1}.HeaderName = 'math.h';
d{2}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{2}.HeaderInfos{2}.HeaderName = 'foo.h';
d{2}.HeaderInfos{2}.ReservedIds = {'c', 'd'};
```

1. ANSI® is a registered trademark of the American National Standards Institute, Inc.
2. ISO® is a registered trademark of the International Organization for Standardization.
3. GNU® is a registered trademark of the Free Software Foundation.

```
d{3}.LibraryName = 'GNU';
d{3}.HeaderInfos{1}.HeaderName = 'math.h';
d{3}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{3}.HeaderInfos{2}.HeaderName = 'foo.h';
d{3}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{4}.LibraryName = 'My Custom TFL';
d{4}.HeaderInfos{1}.HeaderName = 'my_math_lib.h';
d{4}.HeaderInfos{1}.ReservedIds = {'y1', 'u1'};
d{4}.HeaderInfos{2}.HeaderName = 'my_oper_lib.h';
d{4}.HeaderInfos{2}.ReservedIds = {'foo', 'bar'};

setReservedIdentifiers(hLib, d);
```

See Also

"Adding Target Function Library Reserved Identifiers" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set name of server interface

Syntax

```
autosarInterfaceObj.setServerInterfaceName(ServerInterfaceName
    )
```

Description

`autosarInterfaceObj.setServerInterfaceName(ServerInterfaceName)` sets the name of the server interface specified in *`autosarInterfaceObj`*

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`ServerInterfaceName`	Server interface name for *`autosarInterfaceObj`*.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.setServerOperationPrototype

Purpose

Specify operation prototype

Syntax

```
autosarInterfaceObj.setServerOperationPrototype(operation_prot
   otype)
```

Description

`autosarInterfaceObj.setServerOperationPrototype(operation_prototype)` defines the server operation prototype for `autosarInterfaceObj`.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`operation_prototype`

String with names of prototype and arguments:

`operation_name(dir1 datatype1 arg1, dir2 datatype2 arg2, ..., dirN datatypeN argN, ... )`

- `operation_name` — Name of operation
- `dirN` — Either `IN` or `OUT`, which indicates whether data is passed in or out of the function.
- `datatypeN` — Data type, which can be an AUTOSAR basic data type or record, Simulink data type, or array.
- `argN` — Name of the argument

Prototype and argument names must be valid AUTOSAR short-name identifiers.

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set server port name

Syntax

```
autosarInterfaceObj.setServerPortName(serverPortName)
```

Description

`autosarInterfaceObj.setServerPortName(serverPortName)` sets the server port name for the model-specific `RTW.AutosarInterface` object defined by *`autosarInterfaceObj`*.

Input Arguments

`serverPortName`	Name for server port of *`autosarInterfaceObj`*

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

RTW.AutosarInterface.setServerType

Purpose

Specify server type

Syntax

```
autosarInterfaceObj.setServerType(serverType)
```

Description

`autosarInterfaceObj.setServerType(serverType)` specifies whether the server in *`autosarInterfaceObj`* is application software or AUTOSAR Basic Software.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`serverType`	Either `'Application software'` or `'Basic software'`

See Also

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set step method name in model-specific C++ encapsulation interface

Syntax

```
setStepMethodName(obj, fcnName)
```

Description

`setStepMethodName(obj, fcnName)` sets the step method name in the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass`, *obj* = `RTW.ModelCPPVoidClass`, or *obj* = `RTW.getEncapsulationInterfaceSpecification(`*modelName*`)`.
fcnName	String specifying a new name for the step method described by the specified model-specific C++ encapsulation interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the step method name, which you can examine and modify. In the `void-void step method` view, you can examine and modify the step method name without having to click a button. For more information, see "Configuring the Step Method for Your Model Class" in the Real-Time Workshop Embedded Coder documentation.

See Also

"Configuring C++ Encapsulation Interfaces Programmatically" in the Real-Time Workshop Embedded Coder documentation

"Sample Script for Configuring the Step Method for a Model Class" in the Real-Time Workshop Embedded Coder documentation

"Controlling Generation of Encapsulated C++ Model Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set specified parameters for function entry in TFL table

Syntax

```
setTflCFunctionEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a TFL function entry previously returned by *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from `RTW.TflCFunctionEntry`.

varargin

Parameter/value pairs for the function entry. See varargin Parameters.

varargin Parameters

The following function entry parameters can be specified to the `setTflCFunctionEntryParameters` function using parameter/value argument pairs. For example,

```
setTflCFunctionEntryParameters(..., 'Key', 'sqrt', ...);
```

`Key`

String specifying the name of the function to be replaced. The name must match one of the functions supported for replacement:

Math Functions			
abs	cos	log	saturate
acos	cosh	log10	sign
acosh	exactrSqrt	max	sin
asin	exp	min	sinh
asinh	fix	mod/fmod	sqrt
atan	floor	pow	tan
atan2	hypot	rem	tanh
atanh	ldexp	round	

ceil	ln	rSqrt	
Copy Utility Function			
memcpy			
Nonfinite Support Utility Functions			
getInf	getMinusInf	getNaN	

`GenCallback`
: String specifying `''` or `'RTW.copyFileToBuildDir'`. The default is `''`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation.

`Priority`
: Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

`ImplType`
: Specifies the type of entry: `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro. The default is `FCN_IMPL_FUNCT`.

`ImplementationName`
: String specifying the name of the implementation function, for example, `'sqrt'`, which can match or differ from the `Key` name. The default is `''`.

`ImplementationHeaderFile`
: String specifying the name of the header file that declares the implementation function, for example, `'<math.h>'`. The default is `''`.

`ImplementationHeaderPath`
: String specifying the full path to the implementation header file. The default is `''`.

`ImplementationSourceFile`
: String specifying the name of the implementation source file. The default is `''`.

`ImplementationSourcePath`
: String specifying the full path to the implementation source file. The default is `''`.

`AcceptExprInput`
: Boolean value used to flag the code generator that the implementation function described by this entry should accept expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

  If the value is `true`, expression inputs are integrated into the generated code in a form similar to the following:

```
rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

  If the value is `false`, a temporary variable is generated for the expression input, as follows:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

`SideEffects`
: Boolean value used to flag the code generator that the implementation function described by this entry should not be optimized away. This parameter applies to implementation

functions that return `void` but should not be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`. The default is `false`.

Description

The `setTflCFunctionEntryParameters` function sets specified parameters for a function entry in a TFL table.

Example

In the following example, the `setTflCFunctionEntryParameters` function is used to set specified parameters for a TFL function entry for `sqrt`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                 'Key',                     'sqrt', ...
                                 'Priority',                100, ...
                                 'ImplementationName',      'sqrt', ...
                                 'ImplementationHeaderFile', '<math.h>' );
```

See Also

"Example: Mapping Math Functions to Target-Specific Implementations" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set specified parameters for operator entry in TFL table

Syntax

```
setTflCOperationEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by one of the following class instantiations:

`hEntry = RTW.TflCOperationEntry;`	Supports operator replacement, described in "Example: Mapping Scalar Operators to Target-Specific Implementations" and "Mapping Nonscalar Operators to Target-Specific Implementations"
`hEntry = RTW.TflCOperationEntry-Generator;`	Provides relative scaling factor (RSF) fixed-point parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations", that are not available in `RTW.TflCOperationEntry`
`hEntry = RTW.TflCOperationEntry-Generator_NetSlope;`	Provides net slope parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations", that are not available in `RTW.TflCOperationEntry`
`hEntry = RTW.TflBlasEntry-Generator;`	Supports replacement of nonscalar operators with MathWorks BLAS functions, described in "Mapping Nonscalar Operators to Target-Specific Implementations"
`hEntry = RTW.TflCBlasEntry-Generator;`	Supports replacement of nonscalar operators with ANSI/ISO C BLAS functions, described in "Mapping Nonscalar Operators to Target-Specific Implementations"
`hEntry = MyCustomOperationEntry;` (where *MyCustomOperationEntry* is a class derived from `RTW.TflCOperationEntry`)	Supports operator replacement using custom TFL table entries, described in "Refining TFL Matching and Replacement Using Custom TFL Table Entries"

Note If you want to specify any of the parameters `SlopesMustBeTheSame`, `MustHaveZeroNetBias`, `RelativeScalingFactorF`, or `RelativeScalingFactorE` for your operator entry, instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator` rather than *`hEntry`* `= RTW.TflCOperationEntry`. If you want to use `NetSlopeAdjustmentFactor` and `NetFixedExponent`, instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope`.

`varargin`
: Parameter/value pairs for the operator entry. See varargin Parameters.

varargin Parameters

The following operator entry parameters can be specified to the `setTflCOperationEntryParameters` function using parameter/value argument pairs. For example,

```
setTflCOperationEntryParameters(..., 'Key', 'RTW_OP_ADD', ...);
```

`Key`
: String specifying the operator to be replaced, among the operators supported for replacement:

Operator	Key
Addition (+)	`RTW_OP_ADD`
Subtraction (-)	`RTW_OP_MINUS`
Multiplication (*)	`RTW_OP_MUL`
Division (/)	`RTW_OP_DIV`
Data type conversion (cast)	`RTW_OP_CAST`
Shift left (<<)	`RTW_OP_SL`

Operator	Key
Shift right (>>)	RTW_OP_SRA (arithmetic) RTW_OP_SRL (logical)
Complex conjugation	RTW_OP_CONJUGATE
Transposition (.')	RTW_OP_TRANS
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN
Multiplication with transposition	RTW_OP_TRMUL
Multiplication with Hermitian transposition	RTW_OP_HMMUL

The default is 'RTW_OP_ADD'.

`GenCallback`

String specifying '' or 'RTW.copyFileToBuildDir'. The default is ''. If you specify 'RTW.copyFileToBuildDir', and if this operator entry is matched and used, the function RTW.copyFileToBuildDir will be called after code generation to copy additional header, source, or object files that you have specified for this operator entry to the build directory. For more information, see "Specifying Build Information for Function Replacements" in the Real-Time Workshop Embedded Coder documentation.

`Priority`

Positive integer specifying the operator entry's search priority, 0-100, relative to other entries of the same operator name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for an operator, the implementation with the higher priority will shadow the one with the lower priority.

`RoundingMode`
: String specifying the rounding mode supported by the implementation function: `'RTW_ROUND_FLOOR'`, `'RTW_ROUND_CEILING'`, `'RTW_ROUND_ZERO'`, `'RTW_ROUND_NEAREST'`, `'RTW_ROUND_NEAREST_ML'`, `'RTW_ROUND_SIMPLEST'`, `'RTW_ROUND_CONV'`, or `'RTW_ROUND_UNSPECIFIED'`. The default is `'RTW_ROUND_UNSPECIFIED'`.

`SaturationMode`
: String specifying the saturation mode supported by the implementation function: `'RTW_SATURATE_ON_OVERFLOW'`, `'RTW_WRAP_ON_OVERFLOW'`, or `'RTW_SATURATE_UNSPECIFIED'`. The default is `'RTW_SATURATE_UNSPECIFIED'`.

`SlopesMustBeTheSame`
: Boolean flag that, when set to `true`, indicates that TFL replacement request processing must check that the slopes on all arguments (input and output) are equal. The default is `false`.

  This parameter and `MustHaveZeroNetBias` can be used for fixed-point addition and subtraction replacement. Set both parameters to `true` to disregard specific slope and bias values and map relative slope and bias values to a replacement function.

  To use this parameter, you must instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

`MustHaveZeroNetBias`
: Boolean flag that, when set to `true`, indicates that TFL replacement request processing must check that the net bias on all arguments is zero. The default is `false`.

  This parameter and `SlopesMustBeTheSame` can be used for fixed-point addition and subtraction replacement. Set both parameters to `true` to disregard specific slope and bias values and map relative slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.TflCOperationEntryGenerator` rather than `hEntry = RTW.TflCOperationEntry`.

`RelativeScalingFactorF`

Floating-point value specifying the slope adjustment factor (`F`) part of the relative scaling factor, $F2^E$, for relative scaling TFL entries. The default is `1.0`.

This parameter and `RelativeScalingFactorE` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.TflCOperationEntryGenerator` rather than `hEntry = RTW.TflCOperationEntry`.

`RelativeScalingFactorE`

Floating-point value specifying the fixed exponent (`E`) part of the relative scaling factor, $F2^E$, for relative scaling TFL entries. For example, `-3.0`. The default is `0`.

This parameter and `RelativeScalingFactorF` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.TflCOperationEntryGenerator` rather than `hEntry = RTW.TflCOperationEntry`.

`isRSF`

Boolean value specifying that the operator entry is a relative scaling factor (RSF) entry. Specify `true` if the values of `RelativeScalingFactorF` and `RelativeScalingFactorE` equal their defaults, `1.0` and `0`, but the entry nonetheless should be interpreted by the code generation process as an RSF entry.

`NetSlopeAdjustmentFactor`
: Floating-point value specifying the slope adjustment factor (`F`) part of the net slope, $F2^E$, for net slope TFL entries. The default is `1.0`.

  This parameter and `NetFixedExponent` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

  To use this parameter, you must instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

`NetFixedExponent`
: Floating-point value specifying the fixed exponent (`E`) part of the net slope, $F2^E$, for net slope TFL entries. For example, `-3.0`. The default is `0`.

  This parameter and `NetSlopeAdjustmentFactor` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

  To use this parameter, you must instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope` rather than *`hEntry`* `= RTW.TflCOperationEntry`.

`ImplementationName`
: String specifying the name of the implementation function, for example, `'s8_add_s8_s8'`. The default is `''`.

`ImplementationHeaderFile`
: String specifying the name of the header file that declares the implementation function, for example, `'s8_add_s8_s8.h'`. The default is `''`.

`ImplementationHeaderPath`
: String specifying the full path to the implementation header file. The default is `''`.

`ImplementationSourceFile`
: String specifying the name of the implementation source file, for example, `'s8_add_s8_s8.c'`. The default is `''`.

`ImplementationSourcePath`
: String specifying the full path to the implementation source file. The default is `''`.

`AcceptExprInput`
: Boolean value used to flag the code generator that the implementation function described by this entry should accept expression inputs. If the value is `true` (the default), expression inputs are integrated into the generated code in a form similar to the following:

```
rtY.Out1 = u8_add_u8_u8(u8_add_u8_u8(rtU.In1, rtU.In2), rtU.In3);
```

If the value is `false`, a temporary variable is generated for the expression input, as follows:

```
uint8_T tempVar;

tempVar = u8_add_u8_u8(rtU.In1, rtU.In2);
rtY.Out1 = u8_add_u8_u8(tempVar, rtU.In3);
```

`SideEffects`
: Boolean value used to flag the code generator that the implementation function described by this entry should not be optimized away. This parameter applies to implementation functions that return `void` but should not be optimized away, such as an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`. The default is `false`.

Description

The `setTflCOperationEntryParameters` function sets specified parameters for an operator entry in a TFL table.

Example

In the following example, the `setTflCOperationEntryParameters` function is used to set parameters for a TFL operator entry for `uint8` addition.

```
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_UNSPECIFIED', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c' );
```

In the following example, the `setTflCOperationEntryParameters` function is used to set parameters for a TFL operator entry for fixed-point `int16` division. The table entry specifies a relative scaling between the operator inputs and output in order to map a range of slope and bias values to a replacement function.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
                    'Key',                      'RTW_OP_DIV', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_CEILING', ...
                    'RelativeScalingFactorF',   1.0, ...
                    'RelativeScalingFactorE',   -3.0, ...
                    'ImplementationName',       's16_div_s16_s16_rsfOp125', ...
                    'ImplementationHeaderFile', 's16_div_s16_s16_rsfOp125.h', ...
                    'ImplementationSourceFile', 's16_div_s16_s16_rsfOp125.c' );
```

In the following example, the `setTflCOperationEntryParameters` function is used to set parameters for a TFL operator entry for fixed-point `uint16` addition. The table entry specifies equal slope and zero net bias across operator inputs and output in order to map relative slope and bias values (rather than a specific slope and bias combination) to a replacement function.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_WRAP_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'SlopesMustBeTheSame',      true, ...
                    'MustHaveZeroNetBias',      true, ...
                    'ImplementationName',       'u16_add_SameSlopeZeroBias', ...
                    'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
                    'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c' );
```

See Also

"Example: Mapping Scalar Operators to Target-Specific Implementations" in the Real-Time Workshop Embedded Coder documentation

"Mapping Fixed-Point Operators to Target-Specific Implementations" in the Real-Time Workshop Embedded Coder documentation

"Creating Function Replacement Tables" in the Real-Time Workshop Embedded Coder documentation

"Replacing Math Functions and Operators Using Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

slConfigUIGetVal

Purpose

Return current value for custom target configuration option

Syntax

```
value = slConfigUIGetVal(hDlg, hSrc, 'OptionName')
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

Returns

Current value of the specified option. The data type of the return value depends on the data type of the option.

Description

The `slConfigUIGetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUIGetVal` to read the current value of a specified target option.

Example

In the following example, the `slConfigUIGetVal` function returns the value of the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

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

  disp(['Value of IncludeMdlTerminateFcn was ', ...
```

```
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

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

See Also

`slConfigUISetEnabled`, `slConfigUISetVal`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop documentation

"Parameter Command-Line Information Summary" in the Real-Time Workshop documentation

slConfigUISetEnabled

Purpose

Enable or disable custom target configuration option

Syntax

```
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', true)
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', false)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`true`
: Specifies that the option should be enabled.

`false`
: Specifies that the option should be disabled.

Description

The `slConfigUISetEnabled` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetEnabled` to enable or disable a specified target option.

Example

In the following example, the `slConfigUISetEnabled` function disables the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
```

```
      '  Uncheck and disable "Terminate function required".']);

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

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

See Also

`slConfigUIGetVal`, `slConfigUISetVal`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop documentation

"Parameter Command-Line Information Summary" in the Real-Time Workshop documentation

slConfigUISetVal

Purpose

Set value for custom target configuration option

Syntax

```
slConfigUISetVal(hDlg, hSrc, 'OptionName', OptionValue)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`OptionValue`
: Value to be set for the specified option.

Description

The `slConfigUISetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetVal` to set the value of a specified target option.

Example

In the following example, the `slConfigUISetVal` function sets the value `'off'` for the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

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

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

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

See Also

`slConfigUIGetVal`, `slConfigUISetEnabled`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop documentation

"Parameter Command-Line Information Summary" in the Real-Time Workshop documentation

RTW.AutosarInterface.syncWithModel

Purpose

Synchronize configuration with model

Syntax

```
autosarInterfaceObj.syncWithModel
```

Description

`autosarInterfaceObj.syncWithModel` synchronizes the configuration with the model for the `RTW.AutosarInterface` class.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

See Also

"Generating Code for AUTOSAR Software Components" in the Real-Time Workshop Embedded Coder documentation

4

Block Reference

AUTOSAR Client-Server Communication

Invoke AUTOSAR Server Operation	Configure AUTOSAR client port to access Basic Software or application software components
Mode Switch for Invoke AUTOSAR Server Operation	Toggle AUTOSAR client-server operation subsystem blocks between simulation and code generation mode

Configuration Wizards

Custom M-file	Automatically update active configuration parameters of parent model using file containing custom MATLAB code
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Module Packaging

Data Object Wizard	Simulink data object wizard for creating potential Simulink data objects

5

Blocks — Alphabetical List

Custom M-file

Purpose

Automatically update active configuration parameters of parent model using file containing custom MATLAB code

Library

Configuration Wizards

Description

Custom M-file

When you add a Custom M-file block to your Simulink model and double-click it, a custom MATLAB script executes and automatically configures model parameters that are relevant to code generation. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

The MathWorks provides an example MATLAB script, *matlabroot*`/toolbox/rtw/rtw/rtwsampleconfig.m`, that you can use with the Custom M-file block and adapt to your model requirements. The block and the script provide a starting point for customization. For more information, see "Creating a Custom Configuration Wizard Block" in the Real-Time Workshop Embedded Coder documentation.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `Custom` is selected by default.

Configuration function

Name of the predefined or custom MATLAB script to be used to update the active configuration parameters of the parent Simulink model. The default value is `rtwsampleconfig`, which refers to the example script `rtwsampleconfig.m`.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

Data Object Wizard

Purpose

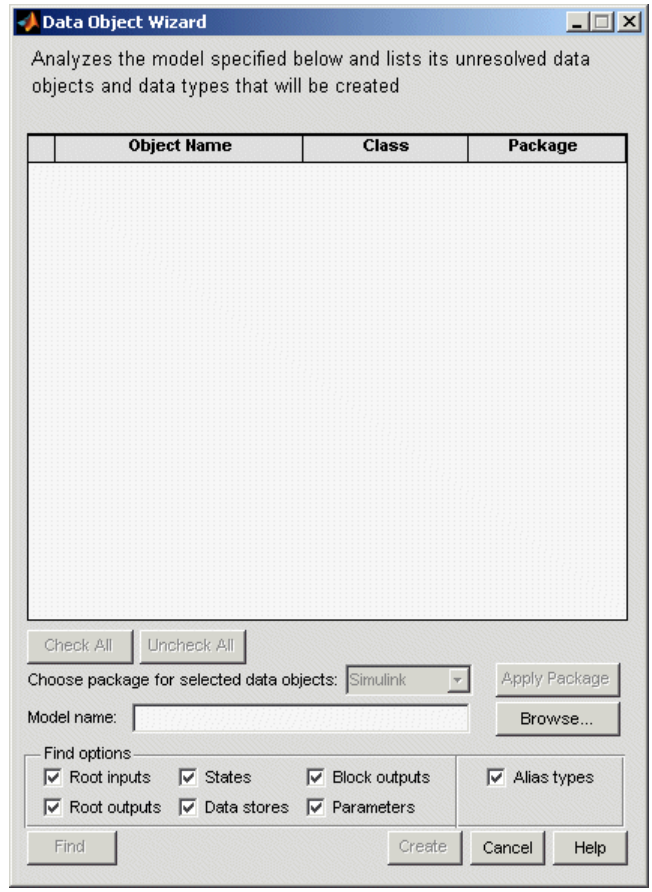
Simulink data object wizard for creating potential Simulink data objects

Library

Module Packaging

Description

When you add a Data Object Wizard block to your Simulink model and double-click it, the Data Object Wizard is launched:



The Data Object Wizard allows you to determine quickly which model data is not associated with Simulink data objects and to create and associate data objects with the data.

For detailed information about using the Data Object Wizard, see "Data Object Wizard" in the Simulink documentation and "Creating Simulink Data Objects with Data Object Wizard" in the Real-Time Workshop Embedded Coder documentation.

You can also launch the Data Object Wizard by entering `dataobjectwizard` at the MATLAB command line or by selecting **Data Object Wizard** from the **Tools** menu of your model.

Example

For an example of a model that incorporates the Data Object Wizard block, see `rtwdemo_mpf`.

See Also

"Data Object Wizard" in the Simulink documentation

"Creating Simulink Data Objects with Data Object Wizard" in the Real-Time Workshop Embedded Coder documentation

"Customizing Data Object Wizard User Packages" in the Real-Time Workshop Embedded Coder documentation

ERT (optimized for fixed-point)

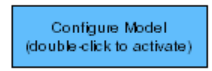
Purpose

Automatically update active configuration parameters of parent model for ERT fixed-point code generation

Library

Configuration Wizards

Description



ERT (optimized for fixed-point)

When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `ERT (optimized for fixed-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for floating-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

ERT (optimized for floating-point)

Purpose

Automatically update active configuration parameters of parent model for ERT floating-point code generation

Library

Configuration Wizards

Description

When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for floating-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `ERT (optimized for floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for fixed-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

GRT (debug for fixed/floating-point)

Purpose

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled

Library

Configuration Wizards

Description

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed/floating-point code generation, with TLC debugging options enabled, with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `GRT (debug for fixed/floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (optimized for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

GRT (optimized for fixed/floating-point)

Purpose

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Library

Configuration Wizards

Description

When you add a GRT (optimized for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed/floating-point code generation with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `GRT (optimized for fixed/floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom M-file block.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

Custom M-file, ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (debug for fixed/floating-point)

"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

Invoke AUTOSAR Server Operation

Purpose

Configure AUTOSAR client port to access Basic Software or application software components

Library

Real-Time Workshop/ AUTOSAR

Description

Use this block to configure an AUTOSAR client port for your Simulink model, which provides access to Basic Software or application software components:

1. Copy or drag this block from the AUTOSAR library into your model.
2. Double-click the block to open the Invoke AUTOSAR Server Operation dialog box.
3. Specify the parameters and click **OK**. This action updates the number of inports and outports to match the operation prototype.
4. Connect this block to other blocks in your model as required.
5. Save and build the model to generate AUTOSAR-compliant code and XML files.

Parameters

Client port name

Must be a valid AUTOSAR short-name identifier.

Operation prototype

Controls the type and number of inports and outports of the block, and must be of the form:

```
operation(prt1 datatype1 arg1, prt2 datatype2 arg2, ...
prtN datatypeN argN, ...)
```

- *`operation`* — Name of the operation
- *`prtN`*. Either `IN` or `OUT`, which indicates whether data passes into or out of the function.

- *datatypeN* — A string indicating data type, which can be an AUTOSAR basic data type or record, Simulink data type, or array.
- *argN* — Name of the argument

Interface path
The reference path for the client-server interface.

Server type
You select the value from:

- `Application software` — For communication with an application software component.
- `Basic software` — For communication with AUTOSAR Basic Software.

For this block, `Application software` is the default.

Show error status
If you select this, client port receives error status of client-server communication.

Sample time (-1 for inherited)
To inherit the sample time, set this parameter to -1.

See Also

Mode Switch for Invoke AUTOSAR Server Operation

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

Mode Switch for Invoke AUTOSAR Server Operation

Purpose

Toggle AUTOSAR client-server operation subsystem blocks between simulation and code generation mode

Library

Real-Time Workshop/ AUTOSAR

Description

You can add this switch block to your Simulink model that contains client-server subsystem blocks. Double-click the switch block to toggle client-server blocks between simulation and code-generation mode.

Parameters

Configure the model for

Value selected from

- `code generation`
- `simulation`

For this block, `code generation` is selected by default.

See Also

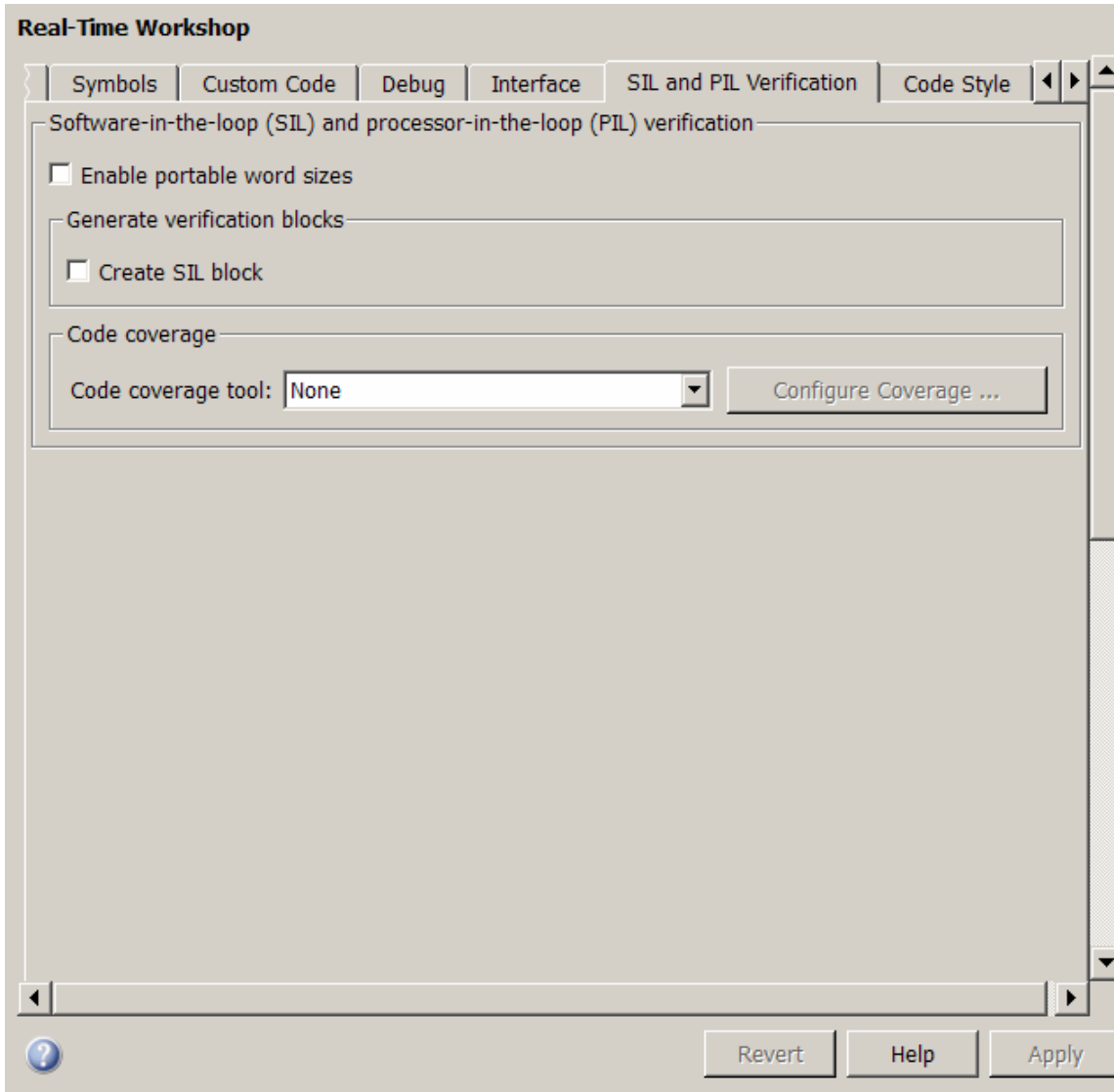
Invoke AUTOSAR Server Operation

"Configuring Client-Server Communication" in the Real-Time Workshop Embedded Coder documentation

6

Configuration Parameters

Real-Time Workshop Pane: SIL and PIL Verification



In this section...

Real-Time Workshop: SIL and PIL Verification Tab Overview

Create SIL block, configure word size portability, and configure code coverage for SIL testing

Configuration

This tab appears only if you specify an ERT–based system target file.

See Also

“Verifying Generated Source Code With Software-in-the-Loop Simulation”

Enable portable word sizes

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

You can enable portable word sizes to support SIL testing of your generated code. To run a SIL simulation, select **Create SIL block** or use top-model or Model block SIL simulation mode.

Settings

Default: off

 On

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

 Off

Does not generate portable code.

Dependencies

When you use this parameter, you should set **Emulation hardware** on the **Hardware Implementation** pane to `None`.

Command-Line Information

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

Recommended Settings

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

See Also

- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes
- Tips for Optimizing the Generated Code
- “Verifying Code Using the SIL Simulation Mode”

Create SIL block

Specify whether to generate a SIL block

Settings

Default: off

 On

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

When this option is selected, the Real-Time Workshop software:

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

 Off

Does not create a SIL block.

Command-Line Information

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

Recommended Settings

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

See Also

- Automatic S-Function Wrapper Generation
- Techniques for Exporting Function-Call Subsystems
- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes
- "Verifying Generated Source Code With Software-In-the-Loop Simulation"

Code coverage tool

Specify a code coverage tool

Settings

Default: `None`

`None`
: No code coverage tool specified

`BullseyeCoverage`
: Specifies the BullseyeCoverage™ tool from Bullseye Testing Technology™

Dependencies

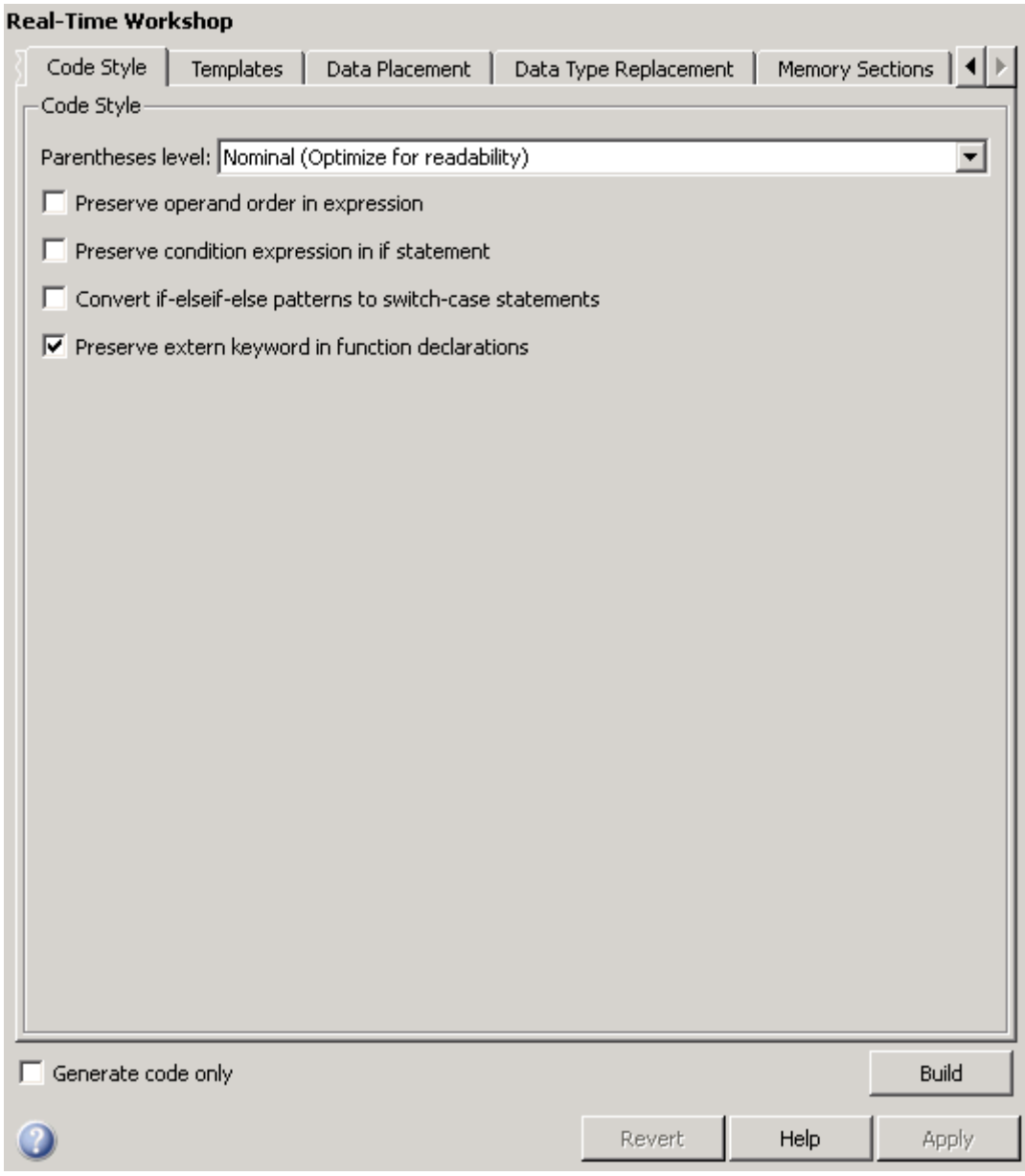
You cannot specify this parameter if the **Create SIL block** check box is selected.

If you do not specify a tool, Configure Coverage appears dimmed. If you specify a tool, click **Configure Coverage** to open the Code Coverage Settings dialog box.

See Also

"Using a Code Coverage Tool in a SIL Simulation"

Real-Time Workshop Pane: Code Style



In this section...

Real-Time Workshop: Code Style Tab Overview

Control optimizations for readability in generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

See Also

- "Controlling Code Style"
- "Real-Time Workshop Pane: Code Style" on page 6-10

Parentheses level

Specify parenthesization style for generated code.

Settings

Default: `Nominal (Optimize for readability)`

`Minimum (Rely on C/C++ operators for precedence)`
: Inserts parentheses only where required by ANSI[4] C or C++, or needed to override default precedence. For example:

```
isZero = var == 0;
if (isZero == 1 && (value < 3.7 ||value > 9.27)) {
   /* code */
   }
```

`Nominal (Optimize for readability)`
: Inserts parentheses in a way that compromises between readability and visual complexity. The exact definition can change between releases.

`Maximum (Specify precedence with parentheses)`
: Includes parentheses everywhere needed to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA®[5] requirements. For example:

```
isZero = (var == 0);
if ((isZero == 1) && ((value < 3.7) || (value > 9.27))) {
   /* code */
   }
```

Command-Line Information

Parameter: `ParenthesesLevel`
Type: string
Value: `'Minimum' | 'Nominal' | 'Maximum'`
Default: `'Nominal'`

4. ANSI® is a registered trademark of the American National Standards Institute, Inc.

5. MISRA® is a registered trademarks of MIRA Ltd, held on behalf of the MISRA® Consortium.

Recommended Settings

Application	Setting
Debugging	`Nominal (Optimized for readability)`
Traceability	`Nominal (Optimized for readability)`
Efficiency	`Minimum (Rely on C/C++ operators for precedence)`
Safety precaution	`Maximum (Specify precedence with parentheses)`

See Also

Controlling Parenthesization

Preserve operand order in expression

Specify whether to preserve order of operands in expressions.

Settings

Default: off

 On

Preserves the expression order specified in the model. Select this option to increase readability of the code or for code traceability purposes.

```
A*(B+C)
```

 Off

Optimizes efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive. For example:

```
(B+C)*A
```

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	On

Preserve condition expression in if statement

Specify whether to preserve empty primary condition expressions in `if` statements.

Settings

Default: off

 On

Preserves empty primary condition expressions in `if` statements, such as the following, to increase the readability of the code or for code traceability purposes.

```
if expression1
else
       statements2;
end
```

 Off

Optimizes empty primary condition expressions in `if` statements by negating them. For example, consider the following `if` statement:

```
if expression1
else
       statements2;
end
```

By default, the code generator negates this statement as follows:

```
if ~expression1
        statements2;
end
```

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	On

Convert if-elseif-else patterns to switch-case statements

Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.

This readability optimization works on a per-model basis and applies only to:

- Flow graphs in Stateflow® charts
- Embedded MATLAB® functions in Stateflow charts
- Embedded MATLAB Function blocks in that model

Settings

Default: off

 On

Generate code for `if-elseif-else` decision logic as `switch-case` statements.

For example, assume that you have the following logic pattern:

```
if (x == 1) {
       y = 1;
} else if (x == 2) {
       y = 2;
} else if (x == 3) {
       y = 3;
} else {
       y = 4;
}
```

Selecting this check box converts the `if-elseif-else` pattern to the following `switch-case` statements:

```
switch (x) {
     case 1:
       y = 1; break;
     case 2:
       y = 2; break;
```

```
        case 3:
          y = 3; break;
        default:
          y = 4; break;
    }
```

 Off

Preserve `if-elseif-else` decision logic in generated code.

Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No impact

See Also

- "Enhancing Readability of Generated Code for Flow Graphs"
- "Enhancing Readability of Generated Code for Embedded MATLAB Function Blocks"
- "Controlling Code Style"

Preserve extern keyword in function declarations

Specify whether to include the `extern` keyword in function declarations in the generated code.

Note The `extern` keyword is optional for functions with external linkage. It is considered good programming practice to include the `extern` keyword in function declarations for code readability.

Settings

Default: on

 On

Include the `extern` keyword in function declarations in the generated code. For example, the generated code for the demo model `rtwdemo_hyperlinks` contains the following function declarations in `rtwdemo_hyperlinks.h`:

```
/* Model entry point functions */
extern void rtwdemo_hyperlinks_initialize(void);
extern void rtwdemo_hyperlinks_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage. The function definitions in this example are in the generated file `rtwdemo_hyperlinks.c`.

 Off

Remove the `extern` keyword from function declarations in the generated code.

Command-Line Information

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

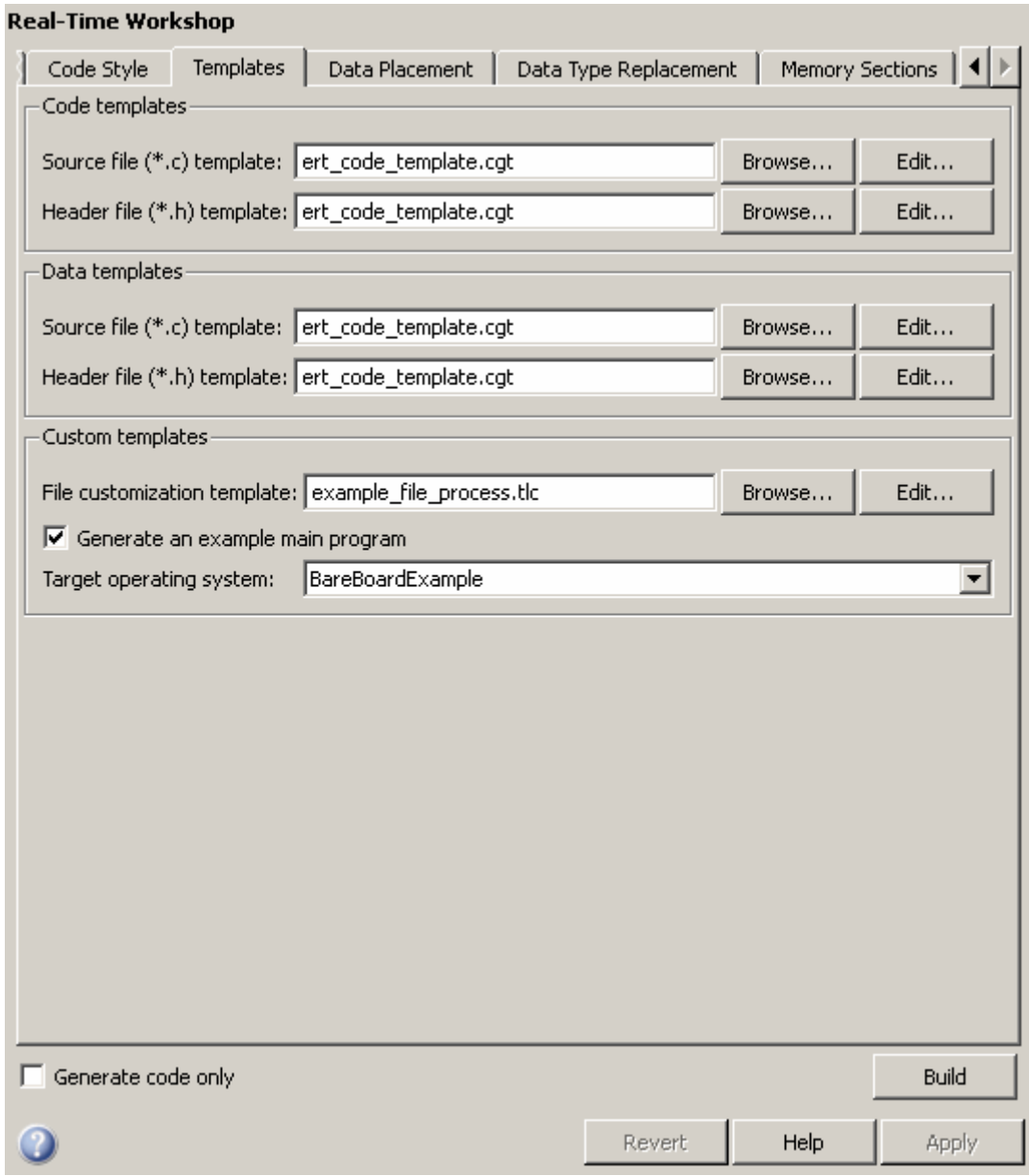
Recommended Settings

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

See Also

For more information on code style options, see "Real-Time Workshop Pane: Code Style" on page 6-10

Real-Time Workshop Pane: Templates



In this section...

Real-Time Workshop: Templates Tab Overview

Customize the organization of your generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

See Also

"Real-Time Workshop Pane: Templates" on page 6-22

Code templates: Source file (*.c) template

Specify the code generation template (CGT) file to use when generating a source code file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated source code files (`.c` or `.cpp`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTSrcFileBannerTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Code templates: Header file (*.h) template

Specify the code generation template (CGT) file to use when generating a code header file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated header files (`.h`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTHdrFileBannerTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Data templates: Source file (*.c) template

Specify the code generation template (CGT) file to use when generating a data source file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data source files (`.c` or `.cpp`) that contain definitions of variables of global scope.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataSrcFileTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Data templates: Header file (*.h) template

Specify the code generation template (CGT) file to use when generating a data header file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data header files (`.h`) that contain declarations of variables of global scope.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataHdrFileTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

File customization template

Specify the custom file processing (CFP) template file to use when generating code.

Settings

Default: `ert_code_template.tlc`

You can use a CFP template file to customize generated code. A CFP template file is a TLC file that organizes types of code (for example, includes, `typedefs`, and functions) into sections. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call a code template API to emit the buffered code into specified sections of generated source and header files. The CFP template file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTCustomFileTemplate`
Type: string
Value: any valid TLC file
Default: `'example_file_process.tlc'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Generate an example main program

Control whether to generate an example main program for a model.

Settings

Default: on

 On

Generates an example main program, `ert_main.c` (or `.cpp`). The file includes:

- The `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

The operation of the main program and the scheduling algorithm employed depend primarily on whether your model is single-rate or multirate, and also on your model's solver mode (`SingleTasking` or `MultiTasking`).

 Off

Provides a static version of the file `ert_main.c` as a basis for custom modifications (*`matlabroot`*`/rtw/c/ert/ert_main.c`). You can use this file as a template for developing embedded applications.

Tips

- After you generate and customize the main program, disable this option to prevent regenerating the main module and overwriting your customized version.
- You can use a custom file processing (CFP) template file to override normal main program generation, and generate a main program module customized for your target environment.
- If you disable this option, the coder generates slightly different rate grouping code to maintain compatibility with an older static `ert_main.c` module.

Dependencies

- This parameter enables **Target operating system**.
- You must enable this parameter and select `VxWorksExample` for **Target operating system** if you use VxWorks®[6] library blocks.

Command-Line Information

Parameter: `GenerateSampleERTMain`
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

- "Generating a Standalone Program"
- Static Main Program Module
- Custom File Processing

6. VxWorks® is a registered trademark of Wind River® Systems, Inc.

Target operating system

Specify a target operating system to use when generating model-specific example main program module.

Settings

Default: `BareBoardExample`

`BareBoardExample`
: Generates a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.

`VxWorksExample`
: Generates a fully commented example showing how to deploy the code under the VxWorks real-time operating system.

Dependencies

- This parameter is enabled by **Generate an example main program**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `TargetOS`
Type: string
Value: `'BareBoardExample' | 'VxWorksExample'`
Default: `'BareBoardExample'`

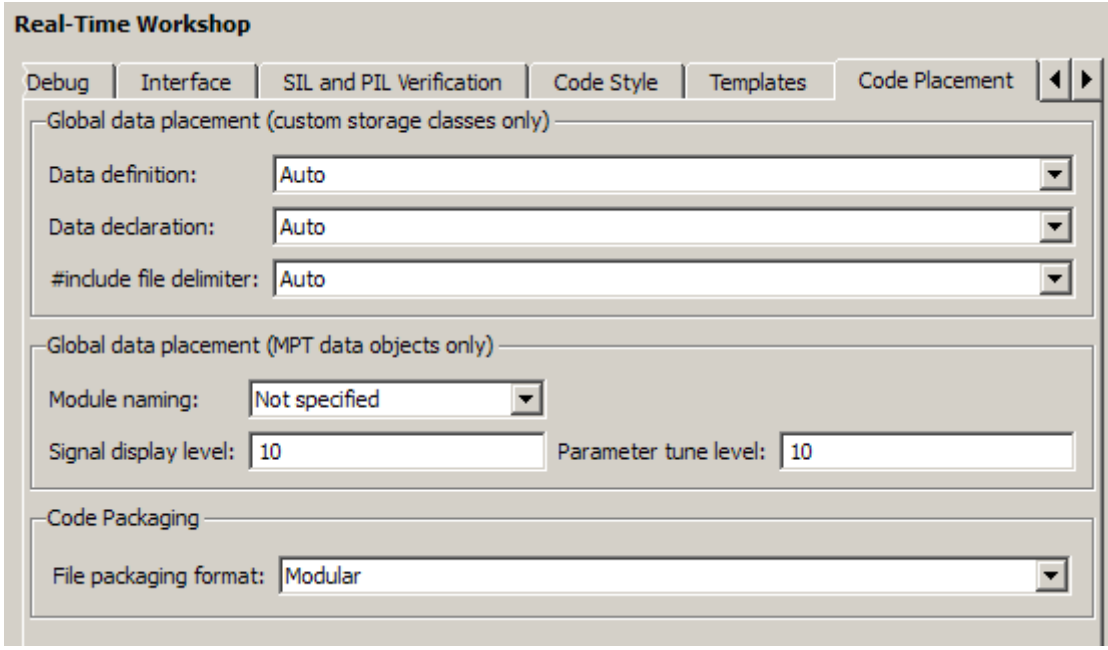
Recommended Settings

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

See Also

- "Generating a Standalone Program"
- Static Main Program Module
- Custom File Processing

Real-Time Workshop Pane: Code Placement



In this section...

In this section...

Real-Time Workshop: Code Placement Tab Overview

Specify the data placement in the generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

See Also

- "Defining Data Representation and Storage for Code Generation"
- "Real-Time Workshop Pane: Code Placement" on page 6-34

Data definition

Specify where to place definitions of global variables.

Settings

Default: `Auto`

`Auto`
: Lets the code generator determine where the definitions should be located.

`Data defined in source file`
: Places definitions in `.c` source files where functions are located. The code generator places the definitions in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`
: Places definitions in the source file specified in the **Data definition filename** field. The code generator organizes and formats the definitions based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Templates** pane.

Dependencies

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data definition filename**.

Command-Line Information

Parameter: `GlobalDataDefinition`
Type: string
Value: `'Auto' | 'InSourceFile' | 'InSeparateSourceFile'`
Default: `'Auto'`

Recommended Settings

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

See Also

- "Overview of Data Placement"
- "Managing Placement of Data Definitions and Declarations"
- "Data Placement Rules and Effects"

Data definition filename

Specify the name of the file that is to contain data definitions.

Settings

Default: `global.c` or `global.cpp`

The code generator organizes and formats the data definitions in the specified file based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Real-Time Workshop** pane: **Templates** tab.

If you specify C++ as the target language, omit the `.cpp` extension. The code generator will generate the correct file and add the extension `.cpp`.

Dependency

This parameter is enabled by **Data definition**.

Command-Line Information

Parameter: `DataDefinitionFile`
Type: string
Value: any valid file
Default: `'global.c'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Data declaration

Specify where `extern`, `typedef`, and `#define` statements are to be declared.

Settings

Default: `Auto`

`Auto`
: Lets the code generator determine where the declarations should be located.

`Data declared in source file`
: Places declarations in `.c` source files where functions are located. The data header template file is not used. The code generator places the declarations in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`
: Places declarations in the data header file specified in the **Data declaration filename** field. The code generator organizes and formats the declarations based on the data header template specified by the **header file (*.h) template** parameter in the data section of the **Real-Time Workshop** pane: **Templates** tab.

Dependencies

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data declaration filename**.

Command-Line Information

Parameter: `GlobalDataReference`
Type: string
Value: `'Auto' | 'InSourceFile' | 'InSeparateHeaderFile'`
Default: `'Auto'`

Recommended Settings

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

See Also

- "Overview of Data Placement"
- "Managing Placement of Data Definitions and Declarations"
- "Data Placement Rules and Effects"

Data declaration filename

Specify the name of the file that is to contain data declarations.

Settings

Default: `global.h`

The code generator organizes and formats the data declarations in the specified file based on the data header template specified by the **Header file (*.h) template** parameter in the data section of the **Real-Time Workshop** pane: **Templates** tab.

Dependency

This parameter is enabled by **Data declaration**.

Command-Line Information

Parameter: `DataReferenceFile`
Type: string
Value: any valid file
Default: `'global.h'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

#include file delimiter

Specify the type of `#include` file delimiter to use in generated code.

Settings

Default: `Auto`

`Auto`
: Lets the code generator choose the `#include` file delimiter

`#include  header.h`
: Uses double quote (" ") characters to delimit file names in `#include` statements.

`#include <header.h>`
: Uses angle brackets (< >) to delimit file names in `#include` statements.

Dependency

The delimiter format that you use when specifying parameter and signal object property values overrides what you set for this parameter.

Command-Line Information

Parameter: `IncludeFileDelimiter`
Type: string
Value: `'Auto' | 'UseQuote' | 'UseBracket'`
Default: `'Auto'`

Recommended Settings

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

Module naming

Specify whether to name the module that owns the model.

Settings

Default: `Not specified`

`Not specified`
: Lets the code generator determine the module name.

`Same as model`
: Uses the name of the model for the module name.

`User specified`
: Uses the module name specified for **Module name** parameter for the module name.

Command-Line Information

Parameter: `ModuleNamingRule`
Type: string
Value: `'Unspecified' | 'SameAsModel' | 'UserSpecified'`
Default: `'Unspecified'`

Dependency

- Selecting `User specified` enables **Module name**.
- Use this parameter with the data object property **Owner** to specify module ownership.
- This parameter must be the same for top-level and referenced models.

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid value

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

- “Overview of Data Placement”
- Ownership Settings

Module name

Specify the name of module that is to own the model.

Settings

Default: `''`

Specify a module name according to ANSI[7] C/C++ conventions for naming identifiers.

Dependency

- This parameter is enabled by `User specified`.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `ModuleName`
Type: string
Value: any valid name
Default: `''`

Recommended Settings

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

See Also

- "Overview of Data Placement"

7. ANSI® is a registered trademark of the American National Standards Institute, Inc.

- Ownership Settings

Signal display level

Specify the persistence level for all MPT signal data objects.

Settings

Default: 10

Specify an integer value indicating the persistence level for all MPT signal data objects. This value indicates the level at which to declare signal data objects as global data in the generated code. The persistence level allows you to make intermediate variables global during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value that you can specify for a specific MPT signal data object in the Model Explorer signal properties dialog.

Dependency

This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SignalDisplayLevel`
Type: integer
Value: any valid integer
Default: `10`

Recommended Settings

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

See Also

Selecting Persistence Level for Signals and Parameters

Parameter tune level

Specify the persistence level for all MPT parameter data objects.

Settings

Default: 10

Specify an integer value indicating the persistence level for all MPT parameter data objects. This value indicates the level at which to declare parameter data objects as tunable global data in the generated code. The persistence level allows you to make intermediate variables global and tunable during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value you that can specify for a specific MPT parameter data object in the Model Explorer parameter properties dialog.

Dependency

This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `ParamTuneLevel`
Type: integer
Value: any valid integer
Default: `10`

Recommended Settings

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

See Also

Selecting Persistence Level for Signals and Parameters

File packaging format

Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. You can specify a different file packaging format for each referenced model.

Settings

Default: `Modular`

`Modular`

- Outputs *`model`*`_data.c`, *`model`*`_private.h`, and *`model`*`_types.h`, in addition to generating *`model`*`.c` and *`model`*`.h`. For the contents of these files, see the table in "Generated Code Modules".
- Supports generating separate source files for subsystems. For more information on generating code for subsystems, see "Creating Subsystems".
- If you specify **Utility function generation** as `Auto` on the **Real-Time Workshop > Interface** pane of the Configuration Parameter dialog box, some utility files are in the build directory. If you specify **Utility function generation** as `Shared location`, separate files are generated for utility functions in a shared location. For more information, see "Controlling Shared Utility Function Generation".

`Compact (with separate data file)`

- Conditionally outputs *`model`*`_data.c`, in addition to generating *`model`*`.c` and *`model`*`.h`.
- If you specify **Utility function generation** as `Auto` on the **Real-Time Workshop > Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *`model`*`.c`. If you specify **Utility function generation** as `Shared location`, separate files are generated for utility functions in a shared location. For more information, see "Controlling Shared Utility Function Generation".
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

`Compact`

- The contents of *`model`*`_data.c` are in *`model`*`.c`.
- The contents of *`model`*`_private.h` and *`model`*`_types.h` are in *`model`*`.h` or *`model`*`.c`.
- If you specify **Utility function generation** as `Auto` on the **Real-Time Workshop > Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *`model`*`.c`. If you specify **Utility function generation** as `Shared location`, separate files are generated for utility functions in a shared location. For more information, see "Controlling Shared Utility Function Generation".
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

Command-Line Information

Parameter: `ERTFilePackagingFormat`
Type: string
Value: `'Modular' | 'CompactWithDataFile' | 'Compact'`
Default: `'Modular'`

Recommended Settings

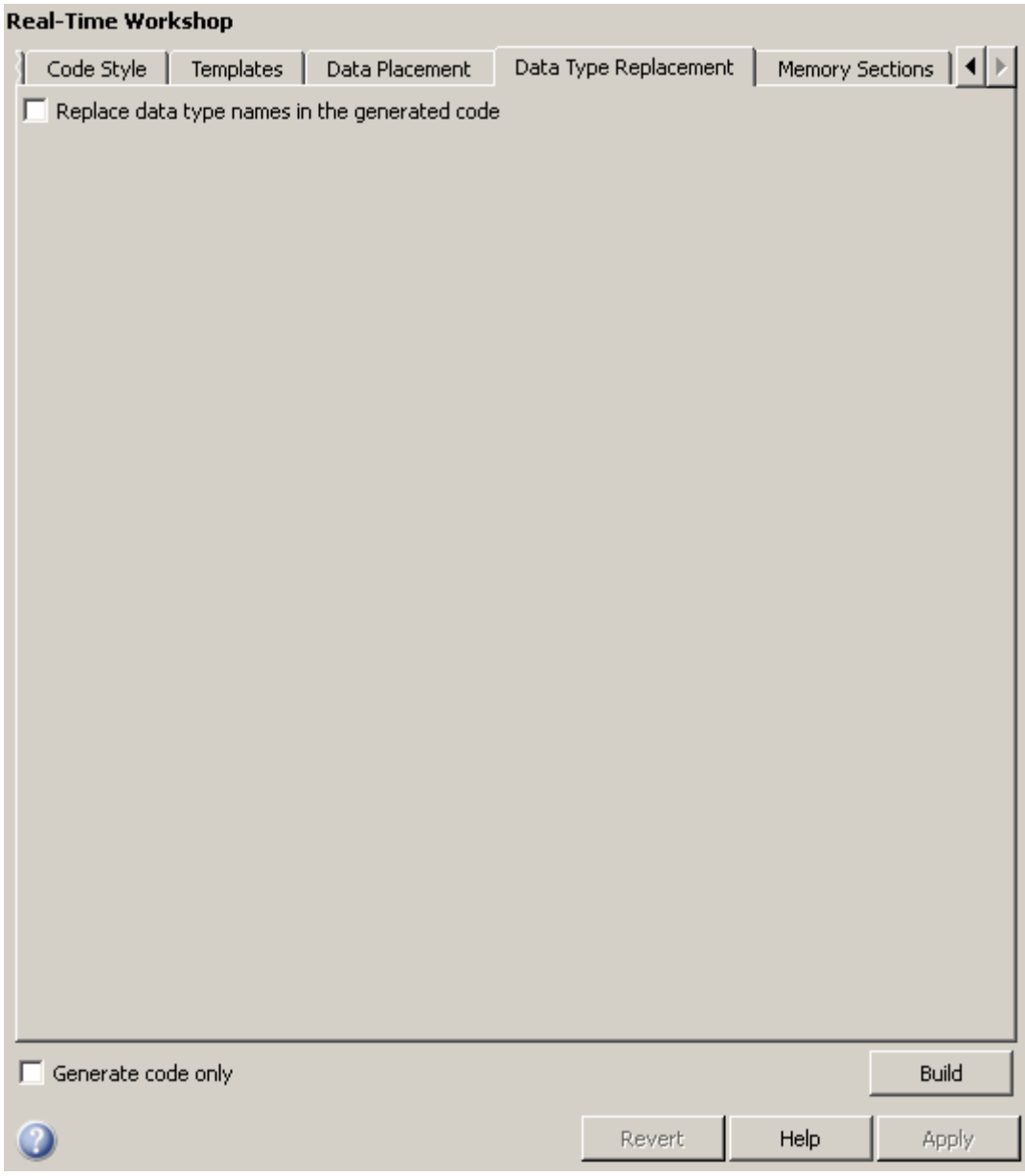
Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Customizing Generated Code Modules "
- "Generating Code Modules"

- “Customizing Post Code Generation Build Processing”

Real-Time Workshop Pane: Data Type Replacement



In this section...

Real-Time Workshop: Data Type Replacement Tab Overview

Replace built-in data type names with user-defined replacement data type names in the generated code for your model.

Configuration

This tab appears only if you specify an ERT based System target file.

If your application requires you to replace built-in data type names with user-defined replacement data type names in the generated code:

1. Select **Replace data type names in the generated code**.
2. Selectively specify replacement data type names to use for built-in Simulink data types in the **Replacement Name** fields.

See Also

- "Replacing Built-In Data Type Names in Generated Code"
- "Real-Time Workshop Pane: Data Type Replacement" on page 6-56

Replace data type names in the generated code

Specify whether to replace built-in data type names with user-defined data type names in generated code.

Settings

Default: off

 On

Displays the **Data type names** table. The table provides a way for you to replace the names of built-in data types used in generated code. This mechanism can be particularly useful for generating code that adheres to application or site data type naming standards.

You can choose to specify new data type names for some or all Simulink built-in data types listed in the table. For each replacement data type name that you specify:

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, and `uint8`, the `BaseType` of the replacement data type must match the built-in data type.
- For `boolean`, the `BaseType` of the replacement data type must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.
- For `int`, `uint`, and `char`, the size of the replacement data type must match the size displayed for **Number of bits: int** or **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

 Off

Uses Real-Time Workshop names for built-in Simulink data types in generated code.

Dependencies

This parameter enables:

double Replacement Name
single Replacement Name
int32 Replacement Name
int16 Replacement Name
int8 Replacement Name
uint32 Replacement Name
uint16 Replacement Name
uint8 Replacement Name
boolean Replacement Name
int Replacement Name
uint Replacement Name
char Replacement Name

Command-Line Information

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

Recommended Settings

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

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: double

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.double`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: single

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.single`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: int32

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int32`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: int16

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types .

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int16`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: int8

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int8`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: uint32

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint32`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: uint16

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint16`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: uint8

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint8`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: boolean

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.boolean`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	`' '`

See Also

- "Replacing `boolean` with an Integer Data Type"
- "Replacing Built-In Data Type Names in Generated Code"

Replacement Name: int

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and the size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

Recommended Settings

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

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: uint

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and the size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Replacement Name: char

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The size of the replacement data type must match the size displayed for on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.char`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and the size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

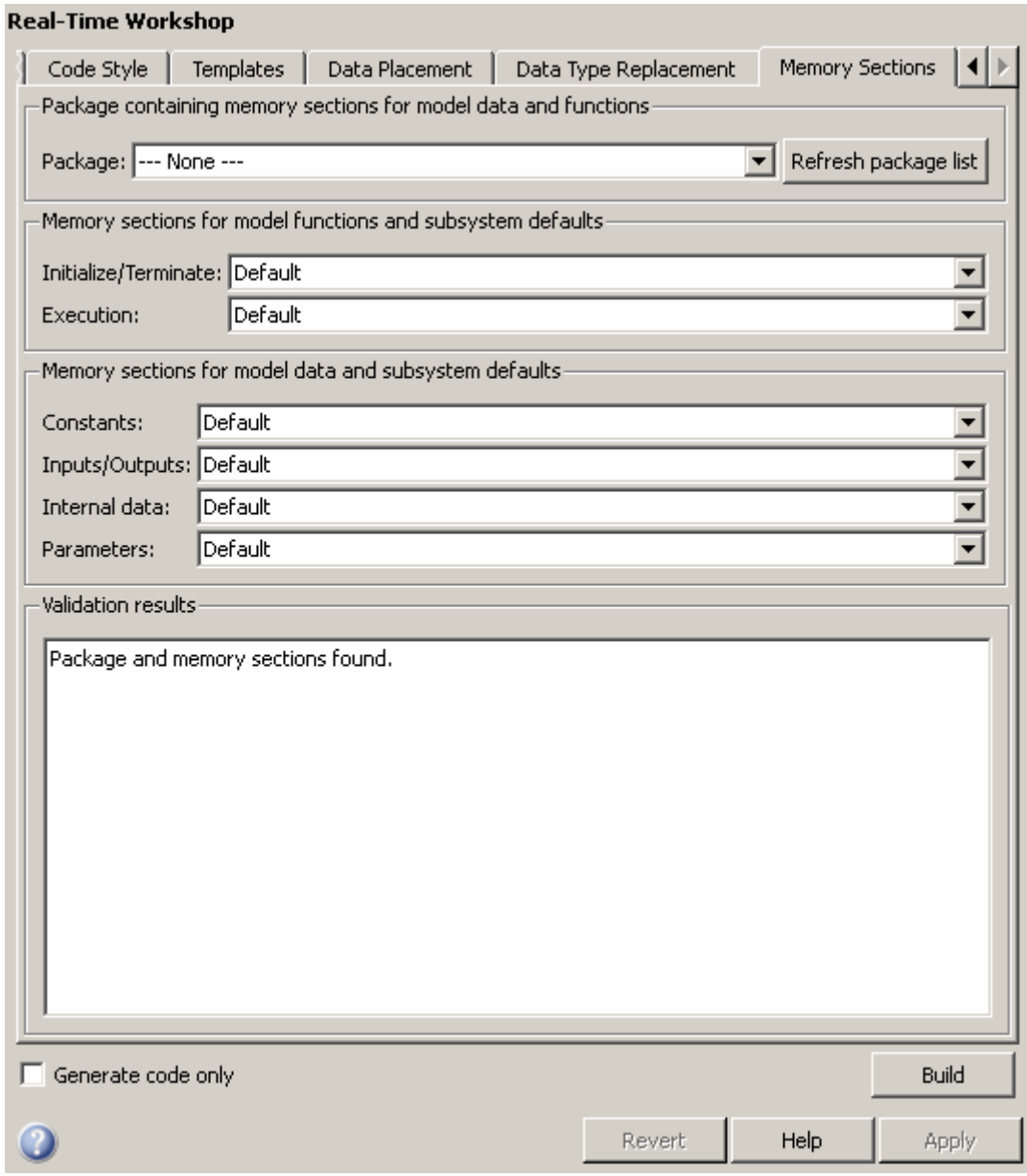
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	' '

See Also

"Replacing Built-In Data Type Names in Generated Code"

Real-Time Workshop Pane: Memory Sections



In this section...

Real-Time Workshop: Memory Sections Tab Overview

Insert comments and pragmas into the generated code for data and functions.

Configuration

This tab appears only if you specify an ERT based system target file.

See Also

- Memory Sections
- “Real-Time Workshop Pane: Memory Sections” on page 6-85

Package

Specify a package that contains memory sections you want to apply to model-level functions and internal data.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `---None---`

`---None---`
: Suppresses memory sections.

`Simulink`
: Applies the built-in Simulink package.

`mpt`
: Applies the built-in mpt package.

Tip

If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

Command-Line Information

Parameter: `MemSecPackage`
Type: string
Value: `'--- None ---' | 'Simulink' | 'mpt'`
Default: `'--- None ---'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Memory Sections

Refresh package list

Add user-defined packages that are on the search path to list of packages displayed by **Packages**.

Tip

If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

See Also

Memory Sections

Initialize/Terminate

Specify whether to apply a memory section to Initialize/Start and Terminate functions.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for Initialize, Start and Terminate functions.

`memory-section-name`
: Applies a memory section to Initialize, Start and Terminate functions.

Command-Line Information

Parameter: `MemSecFuncInitTerm`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

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

See Also

Memory Sections

Execution

Specify whether to apply a memory section to execution functions.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for Step, Run-time initialization, Derivative, Enable, and Disable functions.

`memory-section-name`
: Applies a memory section to Step, Run-time initialization, Derivative, Enable, and Disable functions.

Command-Line Information

Parameter: `MemSecFuncExecute`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

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

See Also

Memory Sections

Constants

Specify whether to apply a memory section to constants.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for constants.

`memory-section-name`
: Applies a memory section to constants.

This parameter applies to:

Data Definition	Data Purpose
`model``_cP`	Constant parameters
`model``_cB`	Constant block I/O
`model``_Z`	Zero representation

Command-Line Information

Parameter: `MemSecDataConstants`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Memory Sections

Inputs/Outputs

Specify whether to apply a memory section to root input and output.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for root-level input and output.

`memory-section-name`
: Applies a memory section for root-level input and output.

This parameter applies to:

Data Definition	Data Purpose
`model``_U`	Root-level input
`model``_Y`	Root-level output

Command-Line Information

Parameter: `MemSecDataIO`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

Memory Sections

Internal data

Specify whether to apply a memory section to internal data.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for internal data.

`memory-section-name`
: Applies a memory section for internal data.

This parameter applies to:

Data Definition	Data Purpose
`model``_B`	Block I/O
`model``_D`	DWork vectors
`model``_M`	Run-time model
`model``_Zero`	Zero-crossings

Command-Line Information

Parameter: `MemSecDataInternal`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

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

See Also

Memory Sections

Parameters

Specify whether to apply a memory section to parameters.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppress the use of a memory section for parameters.

`memory-section-name`
: Apply memory section for parameters.

This parameter applies to:

Data Definition	Data Purpose
`model``_P`	Parameters

Command-Line Information

Parameter: `MemSecDataParameters`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

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

See Also

Memory Sections

Validation results

Display the results of memory section validation.

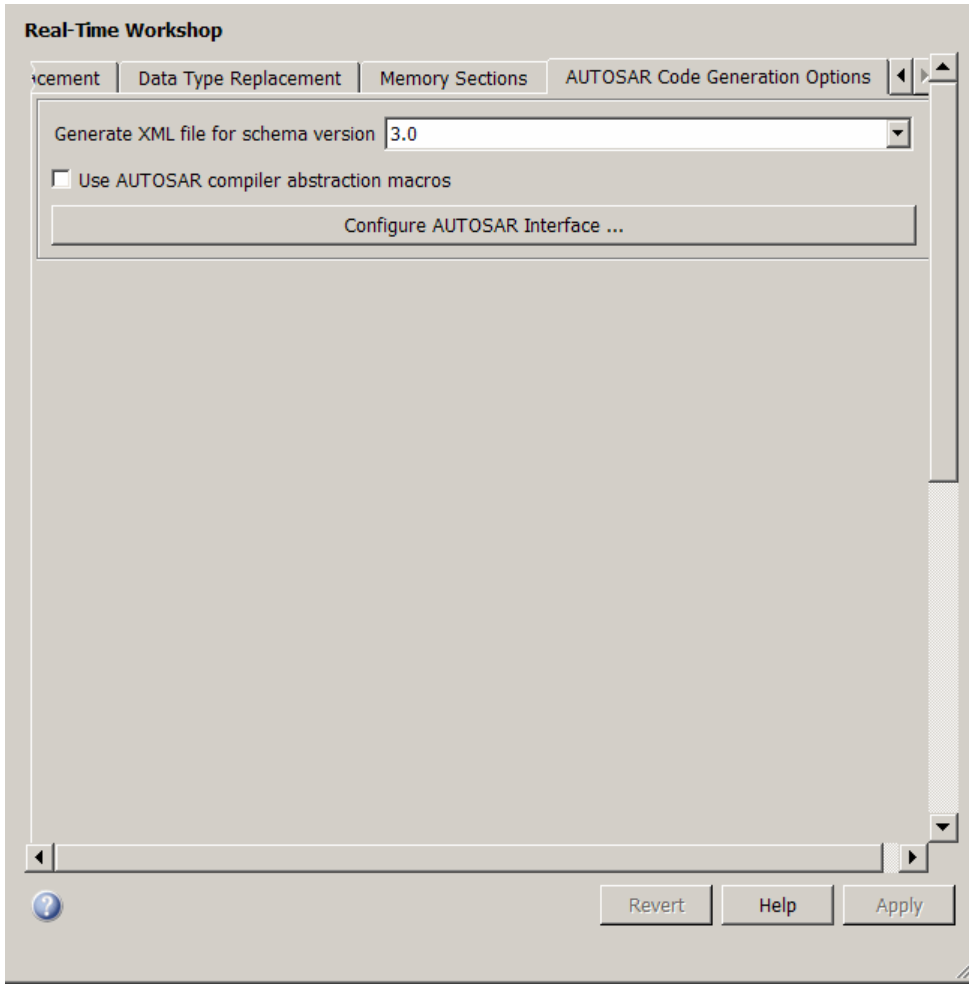
Settings

The Real-Time Workshop software checks and reports whether the currently chosen package is on the MATLAB path and that the selected memory sections exist inside the package.

Recommended Settings

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

Real-Time Workshop Pane: AUTOSAR Code Generation Options



In this section...

Real-Time Workshop: AUTOSAR Code Generation Options Tab Overview

Parameters for controlling AUTOSAR code generation options.

Configuration

This pane appears only if you specify the `autosar.tlc` system target file.

Tip

Click the **Configure AUTOSAR Interface** button to open a dialog box where you can configure all other AUTOSAR options.

See Also

- "Generating Code for AUTOSAR Software Components"
- "AUTOSAR Configuration" on page 1-3
- "AUTOSAR" on page 1-2
- "Real-Time Workshop Pane: AUTOSAR Code Generation Options" on page 6-102

Generate XML file from schema version

Select the AUTOSAR schema version to use when generating XML files.

Settings

Default: `3.0`

`3.0`
: Use schema version 3.0 for XML file generation.

`2.1`
: Use schema version 2.1 for XML file generation.

`2.0`
: Use schema version 2.0 for XML file generation.

Tip

Click the **Configure AUTOSAR Interface** button to open a dialog box where you can configure all other AUTOSAR options.

Command-Line Information

Parameter: `AutosarSchemaVersion`
Type: string
Value: `'3.0' | '2.1' | '2.0'`
Default: `'3.0'`

See Also

"Generating Code for AUTOSAR Software Components"

Use AUTOSAR compiler abstraction macros

Specify use of AUTOSAR macros to abstract compiler directives

Settings

Default: Off

 On

Software generates code with C macros that are abstracted compiler directives (near/far memory calls)

 Off

Software generates code that does *not* contain AUTOSAR compiler abstraction macros.

Command-Line Information

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

See Also

Generating AUTOSAR Compiler Abstraction Macros in the Real-Time Workshop Embedded Coder documentation

Configure AUTOSAR Interface

Opens the Model Interface dialog box where you can configure all other AUTOSAR options.

Dependencies

This parameter is disabled if you are using Configuration Set Reference.

Command-Line Information

Parameter:`autosar_gui_launch`
Type: String
Value: *`subsystemName`*
Default: `No default`

See Also

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Generating Code for AUTOSAR Software Components"

Parameter Reference

In this section...

Recommended Settings Summary

The following table summarizes the impact of each Real-Time Workshop Embedded Coder configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the ERT target. The Real-Time Workshop configuration parameters are documented in "Recommended Settings Summary" in the Real-Time Workshop documentation. For additional details, click the links in the Configuration Parameter column.

Mapping of Application Requirements to the Optimization Pane

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Application lifespan (days)	No impact	No impact	Optimal finite value	`inf`	`1` for ERT targets
Parameter structure	No impact	`Hierarchical`	`Non-Hierarchical`	No impact	`Hierarchical`
Pack Boolean data into bitfields	No impact	No Impact	Off (execution, ROM), On (RAM)	No impact	Off
Bitfield declarator type specifier	No impact	No impact	Target dependent	No impact	`uint_T`
Simplify array indexing	No impact	No impact	No impact	No impact	Off

Mapping of Application Requirements to the Optimization Pane (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Pass reusable subsystem outputs as	No impact	No impact	No impact (execution), `Structure reference` (ROM), `Individual arguments` (RAM)	No impact	`Structure reference`
Remove root level I/O zero initialization	No impact	No impact	On (GUI) `off` (command line) (execution, ROM), No impact (RAM)	Off	Off
Remove internal data zero initialization	No impact	No impact	On (GUI) `off` (command line) (execution, ROM), No impact (RAM)	Off	Off
Optimize initialization code for model reference	No impact	No impact	On (execution, ROM), No impact (RAM)	No impact	On
Remove code that protects against division arithmetic exceptions	No impact	No impact	On	Off	Off

Mapping of Application Requirements to the Real-Time Workshop Pane

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Ignore custom storage classes	No impact	No impact	No impact	No impact	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Report Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Code-to-model	On	On	No impact	On	Off
Model-to-code	On	On	No impact	On	Off
Eliminated / virtual blocks	On	On	No impact	On	Off
Traceable Simulink blocks	On	On	No impact	On	Off
Traceable Stateflow objects	On	On	No impact	On	Off
Traceable Embedded MATLAB functions	On	On	No impact	On	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Comments Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Simulink block descriptions	On	On	No impact	No impact	Off
Simulink data object descriptions	On	On	No impact	No impact	Off
Custom comments (MPT objects only)	On	On	No impact	No impact	Off
Custom comments function	Any valid file name	Any valid file name	No impact	No impact	`''`
Stateflow object descriptions	On	On	No impact	No impact	Off
Requirements in block comments	On	On	No impact	On	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Symbols Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Global variables	No impact	Any valid combination of tokens	No impact	`$R$N$M`	`$R$N$M`
Global types	No impact	Any valid combination of tokens	No impact	`$N$R$M`	`&N$R$M`

Mapping of Application Requirements to the Real-Time Workshop Pane: Symbols Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Field name of global types	No impact	Any valid combination of tokens	No impact	`$N$M`	`$N$M`
Subsystem methods	No impact	Any valid combination of tokens	No impact	`$R$N$M$F`	`$R$N$M$F`
Subsystem method arguments	No impact	Any valid combination of tokens	No impact	`rtu_$N$M` or `rty_$N$M`	`rtu_$N$M` or `rty_$N$M`
Local temporary variables	No impact	Any valid combination of tokens	No impact	`$N$M`	`$N$M`
Local block output variables	No impact	Any valid combination of tokens	No impact	`rtb_$N$M`	`rtb_$N$M`
Constant macros	No impact	Any valid combination of tokens	No impact	`$R$N$M`	`$R$N$M`
Minimum mangle length	No impact	1	No impact	No impact	1
Generate scalar inlined parameters as	No impact	`Macros`	`Literals`	No impact	`Literals`
#define naming	No impact	`Force uppercase`	No impact	No impact	`None`
Parameter naming	No impact	`Force uppercase`	No impact	No impact	`None`

Mapping of Application Requirements to the Real-Time Workshop Pane: Symbols Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Signal naming	No impact	`Force uppercase`	No impact	No impact	`None`
MATLAB function	No impact	No impact	No impact	No impact	`''`

Mapping of Application Requirements to the Real-Time Workshop Pane: Interface Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Support: floating-point numbers	No impact	No impact	Off (GUI), `'on'` (command-line) for integer only	No impact	On (GUI), `'off'` (command-line)
Support complex numbers	No impact	No impact	Off for real only	No impact	On
Support non-finite numbers	No impact	No impact	Off (execution, ROM), No impact (RAM)	Off	On
Support absolute time	No impact	No impact	Off	Off	On
Support continuous time	No impact	No impact	Off (execution, ROM), No impact (RAM)	Off	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Support non-inlined S-functions	No impact	No impact	Off	Off	Off
Support variable-size signals	No impact	No impact	Off	Off	Off
Multiword type definitions	No impact	No impact	Specifying `User defined` and a low value for **Maximum word length** reduces the size of the generated file `rtwtypes.h`	Use default	`System defined`
Maximum word length	No impact	No impact	Smaller values reduce the size of the generated file `rtwtypes.h`	Use default	256
GRT compatible call interface	No impact	Off	Off (execution, ROM), No impact (RAM)	Off	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Single output/update function	On	On	On	On	On
Terminate function required	No impact	No impact	Off (execution, ROM), No impact (RAM)	Off	On
Generate reusable code	No impact	No impact	Set for single instance	No impact	Off
Reusable code error diagnostic	`Warning` or `Error`	No impact	`None`	No impact	`Error`
Pass root-level I/O as	No impact	No impact	No impact	No impact	`Individual arguments`
Block parameter visibility	No impact	No impact	No impact	`protected`	`private`
Internal data visibility	No impact	No impact	No impact	`protected`	`private`
Block parameter access	`Inlined method`	`Inlined method`	`Inlined method`	`None`	`None`
Internal data access	`Inlined method`	`Inlined method`	`Inlined method`	`None`	`None`
External I/O access	`Inlined method`	`Inlined method`	`Inlined method`	`None`	`None`
Generate destructor	No impact	No impact	No impact	Off	On

Mapping of Application Requirements to the Real-Time Workshop Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Use operator new for referenced model object registration	No impact	No impact	On	Off	Off
Generate preprocessor conditionals	No impact	No impact	No impact	No impact	`Use local settings`
Suppress error status in real-time model data structure	Off	No impact	On	On	Off
MAT-file logging	On	No impact	Off	Off	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: SIL and PIL Verification Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Create SIL block"	On	No impact	No impact	No impact	Off
Enable portable word sizes	On	No impact	Off	Off	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Code Style Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Parentheses level	`Nominal (Optimize for readability)`	`Nominal (Optimize for readability)`	`Minimum (Rely on C/C++ operators for precedence)`	`Maximum (Specify precedence with parentheses)`	`Nominal (Optimize for readability)`
Preserve operand order in expression	On	On	Off	On	Off
Preserve condition expression in if statement	On	On	Off	On	Off
Convert if-elseif-else patterns to switch-case statements	No impact	Off	On (execution, ROM), No impact (RAM)	No impact	Off
Preserve extern keyword in function declarations	No impact	No impact	No impact	No impact	On

Mapping of Application Requirements to the Real-Time Workshop Pane: Templates Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Code templates: Source file (*.c) template	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Code templates: Header file (*.h) template	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Data templates: Source file (*.c) template	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
Data templates: Header file (*.h) template	No impact	No impact	No impact	No impact	`ert_code_-template.cgt`
File customization template	No impact	No impact	No impact	No impact	`example_file_-process.tlc`
Generate an example main program	No impact	No impact	No impact	No impact	On
Target operating system	No impact	No impact	No impact	No impact	`BareBoard-Example`

Mapping of Application Requirements to the Real-Time Workshop Pane: Code Placement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Data definition	No impact	Any valid value	No impact	No impact	Auto
Data definition filename	No impact	Any valid value	No impact	No impact	`global.c`
Data declaration	No impact	Any valid value	No impact	No impact	Auto
Data declaration filename	No impact	Any valid value	No impact	No impact	`global.h`
#include file delimiter	No impact	Any valid value	No impact	No impact	Auto
Module naming	No impact	Any valid value	No impact	No impact	Not specified
Module name	No impact	Any valid value	No impact	No impact	`''`
Signal display level	No impact	Any valid integer	No impact	No impact	10
Parameter tune level	No impact	Any valid integer	No impact	No impact	10
File packaging format	No impact	No impact	No impact	No impact	`Modular`

Mapping of Application Requirements to the Real-Time Workshop Pane: Data Type Replacement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Replace data type names in the generated code	No impact	On	No impact	No impact	Off
Replacement Name	No impact	Any valid string	No impact	`''`	`''`

Mapping of Application Requirements to the Real-Time Workshop Pane: Memory Sections Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Package	No impact	No impact	No impact	No impact	`---None---`
Initialize/-Terminate	No impact	No impact	No impact	No impact	`Default`
Execution	No impact	No impact	No impact	No impact	`Default`
Constants	No impact	No impact	No impact	No impact	`Default`
Inputs/Outputs	No impact	No impact	No impact	No impact	`Default`
Internal data	No impact	No impact	No impact	No impact	`Default`
Parameters	No impact	No impact	No impact	No impact	`Default`
Validation results	No impact	No impact	No impact	No impact	Package and memory sections found.

Parameter Command-Line Information Summary

The following tables list Real-Time Workshop Embedded Coder parameters that you can use to tune model and target configurations. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping

to Configuration Parameter dialog box equivalents. For descriptions of the panes and options in that dialog box, see Configuration Parameters in the Real-Time Workshop Embedded Coder documentation.

Use the `get_param` and `set_param` commands to retrieve and set the values of the parameters on the MATLAB command line or programmatically in scripts. The Real-Time Workshop Embedded Coder Configuration Wizard also provides buttons and scripts for customizing code generation.

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

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

Command-Line Information: Optimization Pane

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DataBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bitsets for storing Boolean data**	Use bit sets for storing Boolean data.
`InlinedParameterPlacement` (ERT) **`Hierarchical`**, `NonHierarchical`	**Optimization > Parameter structure**	Specify how generated code stores global (tunable) parameters. Specify `NonHierarchical` to trade off modularity for efficiency.
`BooleansAsBitfields` (ERT) **`off`**, `on`	**Optimization > Pack Boolean data into bitfields**	Specify how generated code stores Boolean signals. If selected, Boolean signals are stored into one-bit bitfields in global block I/O structures or DWork vectors.
`BitfieldContainerType` (ERT) **`uint_T`**, `uchar_T`	**Optimization > Bitfield declarator type specifier**	Specify the bitfield type when using the optimization to pack boolean data into bitfields.
`StrengthReduction` (ERT) **`off`**, `on`	**Optimization > Simplify array indexing**	Suppress generation of code that replaces multiply operations when accessing arrays in a loop.
`PassReuseOutputArgsAs` (ERT) **`Structure reference`**, `Individual arguments`	**Optimization > Pass reusable subsystem output as**	Specify how a reusable subsystem passes outputs. Specify `Individual arguments` for efficiency.
`NoFixptDivByZeroProtection` (ERT) (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 (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`OptimizeModelRefInitCode` (ERT) **`off`**, `on`	**Optimization > Optimize initialization code for model reference**	Suppress generation of initialization code to accommodate the case where this model is referred to by a subsystem that resets its states when enabled. Select this option if the model will never be referred to by such a subsystem. The Simulink engine reports an error if this constraint is violated, in which case you can disable this optimization.
`StateBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bitsets for storing state configuration**	Use bit sets for storing state configuration.
`ZeroExternalMemoryAtStartup` (ERT) `off`, **`on`**	**Optimization > Remove root level I/O zero initialization**	Suppress code that initializes root-level I/O data structures to zero.
`ZeroInternalMemoryAtStartup` (ERT) `off`, **`on`**	**Optimization > Remove internal data zero initialization**	Suppress code that initializes global data structures (for example, block I/O data structures) to zero.

Command-Line Information: Real-Time Workshop Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IgnoreCustomStorageClasses` (ERT) *`string`* - `off`, **`on`**	**Real-Time Workshop > General > Ignore custom storage classes**	Treat custom storage classes as `'Auto'`.

Command-Line Information: Real-Time Workshop Pane: Report Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateTraceInfo` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Report > Model-to-code**	Includes model-to-code traceability support in the generated HTML report.
`IncludeHyperlinkInReport` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Report > Code-to-model**	Link code segments to the corresponding object in the model. This option increases code generation time for large models.
`GenerateTraceReport` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Report > Eliminated / virtual blocks**	Include summary of eliminated and virtual blocks in Code Generation report.
`GenerateTraceReportSl` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Report > Traceable Simulink blocks**	Include summary of Simulink blocks in Code Generation report.
`GenerateTraceReportSf` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Report > Traceable Stateflow objects**	Include summary of Stateflow objects in Code Generation report.
`GenerateTraceReportEml` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Report > Traceable Embedded MATLAB functions**	Include summary of Embedded MATLAB functions in Code Generation report.

Command-Line Information: Real-Time Workshop Pane: Comments Tab

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

Command-Line Information: Real-Time Workshop Pane: Symbols Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrBlkIO` (ERT) *`string`* - **`rtb_$N$M`**	**Real-Time Workshop > Symbols > Local block output variables**	Specify a symbol format rule for local block output variables. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$N` - Name of object `$A` - Data type acronym
`CustomSymbolStrFcn` (ERT) *`string`* - **`$R$N$M$F`**	**Real-Time Workshop > Symbols > Subsystem methods**	Specify a symbol format rule for subsystem methods. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$R` - Root model name `$N` - Name of object `$H` - System hierarchy number `$F` - Subsystem method name
`CustomSymbolStrFcnArg`(ERT) *`string`* - **`rtu_$N$M`** or **`rty_$N$M`**	**Real-Time Workshop > Symbols > Subsystem method arguments**	Specify a symbol format rule for subsystem method arguments. The rule can contain valid C identifier characters and the following macros: `$I` — `u` if the argument is an input or `y` if the argument is an output `$M` - Mangle `$N` - Name of object

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

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

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrTmpVar` (ERT) *`string`* - **`$N$M`**	**Real-Time Workshop > Symbols > Local temporary variables**	Specify a symbol format rule for local temporary variables. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$R` - Root model name `$N` - Name of object
`CustomSymbolStrType` (ERT) *`string`* - **`$N$R$M`**	**Real-Time Workshop > Symbols > Global types**	Specify a symbol format rule for global types. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$R` - Root model name `$N` - Name of object
`DefineNamingFcn` (ERT) *`string`* -	**Real-Time Workshop > Symbols > #define naming > Custom M-function**	Specify a custom MATLAB function to control the naming of symbols with `#define` statements. You can set this parameter only if `DefineNamingRule` is set to `Custom`.
`DefineNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > #define naming**	Specify the rule that changes the spelling of all `#define` names.

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncDataTypeInIds` (ERT) **`off`**, `on`	**Real-Time Workshop > Symbol > Include data type acronym in identifiers**	Include acronyms that express data types in signal and work vector identifiers. For example, `'rtB.i32_signame'` identifies a 32-bit integer block output signal named `'signame'`.
`IncHierarchyInIds` (ERT) **`off`**, `on`	**Real-Time Workshop > Symbols > Include system hierarchy number in identifiers**	Include the system hierarchy number in variable identifiers. For example, `'s3_'` is the system hierarchy number in `rtB.s3_signame` for a block output signal named `'signame'`. Including the system hierarchy number in identifiers improves the traceability of generated code. To locate the subsystem in which the identifier resides, type `hilite_system('<S3>')` at the MATLAB prompt. The argument specified with `hilite_system` requires an uppercase `S`.
`InlinedPrmAccess` (ERT) string - **`Literals`**, `Macros`	**Real-Time Workshop > Symbols > Generate scalar inlined parameters as**	Specify whether inlined parameters are coded as numeric constants or macros. Specify `Macros` for more efficient code.

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MangleLength` (ERT) `int - 1`	**Real-Time Workshop > Symbols > Minimum mangle length**	Specify the minimum number of characters to be used for name mangling strings generated and applied to symbols to avoid name collisions. A larger value reduces the chance of identifier disturbance when you modify the model.
`ParamNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > Parameter naming**	Select a rule that changes spelling of all parameter names.
`PrefixModelToSubsysFcnNames` (ERT) `off`, **`on`**	**Real-Time Workshop > Symbols > Prefix model name to global identifiers**	Add the model name as a prefix to subsystem function names for all code formats. When appropriate for the code format, also add the model name as a prefix to top-level functions and data structures. This prevents compiler errors due to name clashes when combining multiple models.
`SignalNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > Signal naming**	Specify a rule the code generator is to use that changes spelling of all signal names.

Command-Line Information: Real-Time Workshop Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CombineOutputUpdateFcns` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Single output/update function**	Generate a model's output and update routines into a single-step function.
`ERTMaxMultiwordLength` (ERT) `int` - **`256`**	**Real-Time Workshop > Interface > Maximum word length**	Specify a maximum word length, in bits, for which the code generation process will generate system-defined multiword types into the file `rtwtypes.h`. Specifying 0 provides you complete control over type definitions for multiword data types in generated code.
`ERTMultiwordTypeDef` (ERT) string - **`System defined`**, `User defined`	**Real-Time Workshop > Interface > Multiword type definitions**	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
`GenerateDestructor` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Generate destructor**	Generate a destructor for the model class in `C++ (Encapsulated)` model code.
`GenerateExternalIOAccess-Methods` (ERT) string - **`None`**, `Method`, `Inlined method`	**Real-Time Workshop > Interface > External I/O access**	Specify whether to generate access methods for root-level I/O signals for the `C++ (Encapsulated)` model class.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateInternalMember-AccessMethods` (ERT) string - **`None`**, `Method`, `Inlined method`	**Real-Time Workshop > Interface > 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++ (Encapsulated)` model class.
`GenerateParameterAccess-Methods` (ERT) string - **`None`**, `Method`, `Inlined method`	**Real-Time Workshop > Interface > Block parameter access**	Specify whether to generate access methods for block parameters for the `C++ (Encapsulated)` model class.
`GeneratePreprocessor-Conditionals` (ERT) string - **`Use local settings`**, `Enable all`, `Disable all`	**Real-Time Workshop > Interface > Generate preprocessor conditionals**	Specify whether to generate preprocessor conditionals locally for each Model block containing variants or globally for all Model blocks in a model.
`GRTInterface` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > GRT compatible call interface**	Include a code interface (wrapper) that is compatible with the GRT target.
`IncludeMdlTerminateFcn` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Terminate function required**	Generate a terminate function for the model.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`InternalMemberVisibility` (ERT) string - `public`, **`private`**, `protected`	**Real-Time Workshop > Interface > 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++ (Encapsulated)` model class.
`MatFileLogging` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > MAT-file logging**	Generate code that logs data to a MAT-file.
`MultiInstanceErrorCode` (ERT) string - `None`, `Warning`, **`Error`**	**Real-Time Workshop > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.
`MultiInstanceERTCode` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ParameterMemberVisibility` (ERT) string - `public`, **`private`**, `protected`	**Real-Time Workshop > Interface > Block parameter visibility**	Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the `C++ (Encapsulated)` model class.
`PurelyIntegerCode` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > floating-point numbers**	Support floating-point data types in the generated code. This option is forced on when `SupportNonInlinedSFcns` is on.
`RootIOFormat` (ERT) string - **`Individual arguments`**, `Structure reference`	**Real-Time Workshop > Interface > Pass root-level I/O as**	Specify how the code generator is to pass root-level I/O data into a reusable function.
`SupportAbsoluteTime` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > absolute time**	Support absolute time in the generated code. Blocks such as the Discrete Integrator might require absolute time.
`SupportComplex` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > complex numbers**	Support complex data types in the generated code.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SupportContinuousTime` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > continuous time**	Support continuous time in the generated code. This allows blocks to be configured with a continuous sample time. Not available if `SuppressErrorStatus` is `on`.
`SupportNonFinite` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > nonfinite numbers**	Support nonfinite values (`inf`, `nan`, `-inf`) in the generated code. This option is forced on when `SupportNonInlinedSFcns` is on.
`SupportVariableSizeSignals` (ERT) string - **`off`**, `on`	**Real-Time Workshop >Interface > variable-size signals**	Specify whether to generate code for models that use variable-size signals.
`SuppressErrorStatus` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Suppress error status in real-time model data structure**	Remove the error status field of the real-time model data structure to preserve memory. When selected, `SupportContinuousTime` is cleared.
`UseOperatorNewForModelRef-Registration` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Use operator new for referenced model object registration**	For a model containing Model blocks, specify whether generated code should use the operator `new`, during model object registration, to instantiate objects for referenced models configured with

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		a C++ encapsulation interface.

Command-Line Information: Real-Time Workshop Pane: SIL and PIL Verification Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateErtSFunction` (ERT) string - **`off`**, `on`	**Real-Time Workshop > SIL and PIL Verification > Create SIL block**	Wrap the generated code inside an S-Function block. This allows you to validate the generated code in a Simulink model.
`PortableWordSizes` (ERT) string - **`off`**, `on`	**Real-Time Workshop > SIL and PIL Verification > Enable portable word sizes**	Specify that model code should be generated with conditional processing macros that allow the same generated source code files to be used both for software-in-the-loop (SIL) testing on the host platform and for production deployment on the target platform.

Command-Line Information: Real-Time Workshop Pane: Code Style Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ConvertIfToSwitch` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Convert if-elseif-else patterns to switch-case statements**	Control whether `if-elseif-else` decision logic appears in generated code as `switch-case` statements.
`ParenthesesLevel` (ERT) string - `Minimum`, **`Nominal`**, `Maximum`	**Real-Time Workshop > Code Style > Parentheses Level**	Control existence of optional parentheses in generated code.
`PreserveExpressionOrder` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve operand order in expression**	Control reordering of commutable expressions.
`PreserveExternInFcnDecls` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Code Style > Preserve extern keyword in function declarations**	Control whether `extern` keyword appears in function declarations with external linkage in the generated code.
`PreserveIfCondition` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve condition expression in if statement**	Control preservation of `if` statement conditions.

Command-Line Information: Real-Time Workshop Pane: Templates Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ERTCustomFileTemplate` (ERT) *`string`* - **`example_file_process.tlc`**	**Real-Time Workshop > Templates > File customization template**	Specify a TLC callback script for customizing the generated code.
`ERTDataHdrFileTemplate` (ERT) *`string`* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Header file (*.h) template**	Specify a template that organizes the generated data `.h` header files.

Command-Line Information: Real-Time Workshop Pane: Templates Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ERTDataSrcFileTemplate` (ERT) *string* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Source file (*.c or *.cpp) template**	Specify a template that organizes the generated data `.c` source files.
`ERTHdrFileBannerTemplate` (ERT) *string* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Header file (*.h) template**	Specify a template that organizes the generated code `.h` header files.
`ERTSrcFileBannerTemplate` (ERT) *string* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Source file (*.c or *.cpp) template**	Specify a template that organizes the generated code `.c` or `.cpp` source files.
`GenerateSampleERTMain` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Templates > Generate an example main program**	Generate an example main program that demonstrates how to deploy the generated code. The program is written to the file `ert_main.c` or `ert_main.cpp`.
`TargetOS` (ERT) string - **`BareBoardExample`**, `VxWorksExample`	**Real-Time Workshop > Templates > Target operating system**	Specify the target operating system for the example main `ert_main.c` or `ert_main.cpp`. `BareBoardExample` is a generic example that assumes no operating system. `VxWorksExample` is tailored to the VxWorks[8] real-time operating system.

8. VxWorks® is a registered trademark of Wind River® Systems, Inc.

Command-Line Information: Real-Time Workshop Pane: Code Placement Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DataDefinitionFile` (ERT) *`string`* - **`global.c`**	**Real-Time Workshop > Code Placement > Data definition filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data definitions.
`DataReferenceFile` (ERT) *`string`* - **`global.h`**	**Real-Time Workshop > Code Placement > Data declaration filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data references.
`GlobalDataDefinition` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateSourceFile`	**Real-Time Workshop > Code Placement > Data definition**	Select the `.c` or `.cpp` file where variables of global scope are defined.
`GlobalDataReference` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateHeaderFile`	**Real-Time Workshop > Data Placement > Data declaration**	Select the `.h` file where variables of global scope are declared (for example, `extern real_T globalvar;`).
`IncludeFileDelimiter` (ERT) string - **`Auto`**, `UseQuote`, `UseBracket`	**Real-Time Workshop > Code Placement > #include file delimiter**	Specify the delimiter to be used for all data objects that do not have a delimiter specified in the `IncludeFile` property.
`ModuleName` (ERT) *`string`* -	**Real-Time Workshop > Code Placement > Module name**	Specify the name of the module that owns this model.
`ModuleNamingRule` (ERT) string - **`Unspecified`**, `SameAsModel`, `UserSpecified`	**Real-Time Workshop > Code Placement > Module naming**	Specify the rule to be used for naming the module.
`ParamTuneLevel` (ERT) `int` - **`10`**	**Real-Time Workshop > Code Placement > Parameter tune level**	Specify whether the code generator is to declare a parameter data object as tunable global data in the generated code.

Command-Line Information: Real-Time Workshop Pane: Code Placement Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SignalDisplayLevel` (ERT) `int` - **`10`**	**Real-Time Workshop > Code Placement > Signal display level**	Specify whether the code generator is to declare a signal data object as global data in the generated code.
`ERTFilePackagingFormat` (ERT) string - **`Modular`**, `Compact` `with separate data files`, `Compact`	**Real-Time Workshop > Code Placement > File Packaging Format**	Specify how the code generator organizes the code into files.

Command-Line Information: Real-Time Workshop Pane: Data Type Replacement Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`EnableUserReplacementTypes` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Data Type Replacement**	Specify whether to replace built-in data type names with user-defined data type names in generated code.
`ReplacementTypes` (ERT) *`string`* -	**Real-Time Workshop > Data Type Replacement > Data type names**	Specify names to use for built-in data types in generated code.

Command-Line Information: Real-Time Workshop Pane: Memory Sections Tab

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

Command-Line Information: Real-Time Workshop Pane: Memory Sections Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MemSecDataInternal` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Internal data**	Apply memory sections to: • Block I/O • DWork vectors • Run-time model • Zero-crossings
`MemSecDataParameters` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Real-Time Workshop > Memory Sections > Parameters**	Apply memory sections to: • Parameters

Command-Line Information: Not in GUI

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CPPClassGenCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to generate and configure C++ encapsulation interfaces to model code. Default is `off` for custom and non-ERT targets and `on` for ERT (`ert.tlc`) targets. (For more information, see "Supporting C++ Encapsulation Interface Control" in the Real-Time Workshop documentation.)
`ERTFirstTimeCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		to control inclusion of the `firstTime` argument in the *`model`*`_initialize` function generated for a Simulink model. Default is `off` for custom and non-ERT targets and `on` for ERT targets. (For more information, see "Supporting firstTime Argument Control" in the Real-Time Workshop documentation.)
`IncludeERTFirstTime` (ERT) string - **`off`**, `on` **Note** The value of `IncludeERTFirstTime` is meaningful only if the target configuration parameter `ERTFirstTimeCompliant` is set to `on` for your selected target.	Not available	Specify whether Real-Time Workshop Embedded Coder software is to include the `firstTime` argument in the *`model`*`_initialize` function generated for a Simulink model.
`ModelStepFunctionPrototype-ControlCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to control the function prototypes of initialize and step functions that are generated for a Simulink model. Default is `off` for non-ERT targets and `on` for ERT targets. (For more information, see "Supporting C Function Prototype Control"

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		in the Real-Time Workshop documentation.)

Index

A

B

C

D

E

F

G

I

M

P

R

S

T