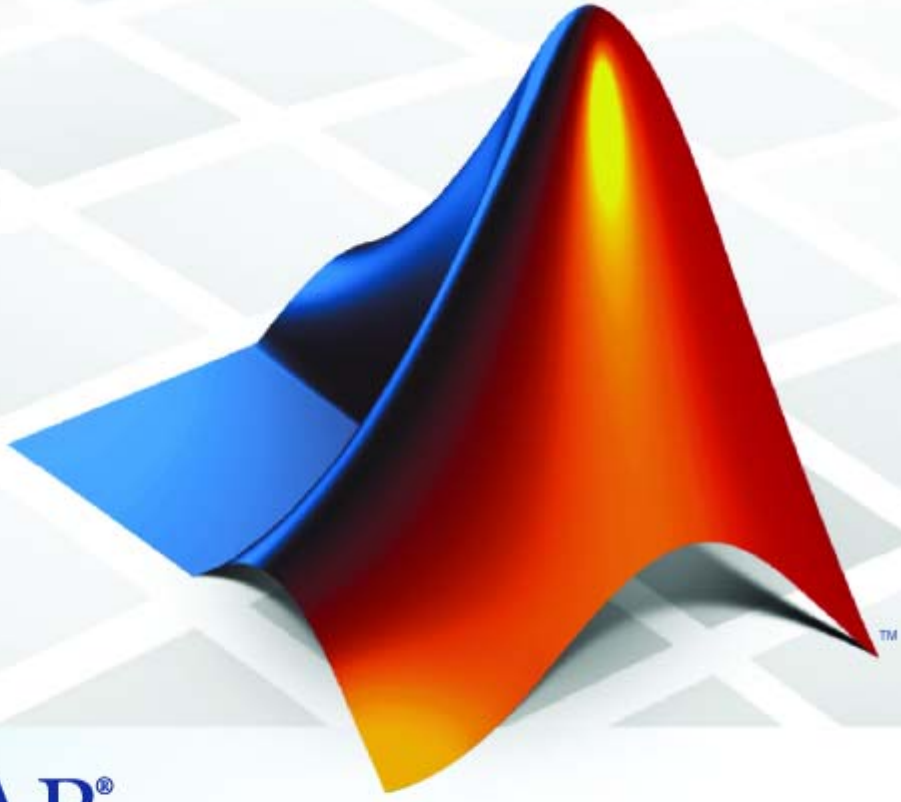
Real-Time Workshop® Embedded Coder™ 5
Reference



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)

Contents

1 Function Reference

2 Functions — Alphabetical List

3 Block Reference

4 Blocks — Alphabetical List

5 Configuration Parameters

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

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

AUTOSAR Configuration

`addIOConf`	Add AUTOSAR I/O configuration to a model
`attachToModel (AUTOSAR)`	Attach `RTW.AutosarInterface` object to model
`getComponentName`	Get XML component name
`getDataTypePackageName`	Get XML data type package name
`getDefaultConf (AUTOSAR)`	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 name
`getIOServiceInterface`	Get port I/O service interface
`getIOServiceName`	Get port I/O service name
`getIOServiceOperation`	Get port I/O service operation
`getPeriodicEventName`	Get periodic event name
`getPeriodicRunnableName`	Get periodic runnable name
`getPortDefaultConf`	Get port default configuration
`runValidation (AUTOSAR)`	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
`setPeriodicEventName`	Set periodic event name
`setPeriodicRunnableName`	Set periodic runnable name
`syncWithModel`	Synchronize configuration with model

C++ Encapsulation Interface Control

`attachToModel (C++ Encapsulation Interface Control)`	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
`getArgCategory (C++ Encapsulation Interface Control)`	Get argument category for Simulink model port from model-specific C++ encapsulation interface
`getArgName (C++ Encapsulation Interface Control)`	Get argument name for Simulink model port from model-specific C++ encapsulation interface
`getArgPosition (C++ Encapsulation Interface Control)`	Get argument position for Simulink model port from model-specific C++ encapsulation interface
`getArgQualifier (C++ Encapsulation Interface Control)`	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface

`getClassName`	Get class name from model-specific C++ encapsulation interface
`getDefaultConf (C++ Encapsulation Interface Control)`	Get default configuration information for model-specific C++ encapsulation interface from Simulink model to which it is attached
`getNumArgs (C++ Encapsulation Interface Control)`	Get number of step method arguments from model-specific C++ encapsulation interface
`getStepMethodName`	Get step method name from model-specific C++ encapsulation interface
`RTW.configSubsystemBuild`	Open GUI to 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
`runValidation (C++ Encapsulation Interface Control)`	Validate model-specific C++ encapsulation interface against Simulink model to which it is attached
`setArgCategory (C++ Encapsulation Interface Control)`	Set argument category for Simulink model port in model-specific C++ encapsulation interface
`setArgName (C++ Encapsulation Interface Control)`	Set argument name for Simulink model port in model-specific C++ encapsulation interface
`setArgPosition (C++ Encapsulation Interface Control)`	Set argument position for Simulink model port in model-specific C++ encapsulation interface

`setArgQualifier (C++ Encapsulation Interface Control)`	Set argument type qualifier for Simulink model port in 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

Function Prototype Control

`addArgConf`	Add argument configuration information for Simulink model port to model-specific C function prototype
`attachToModel (Function Prototype Control)`	Attach model-specific C function prototype to loaded ERT-based Simulink model
`getArgCategory (Function Prototype Control)`	Get argument category for Simulink model port from model-specific C function prototype
`getArgName (Function Prototype Control)`	Get argument name for Simulink model port from model-specific C function prototype
`getArgPosition (Function Prototype Control)`	Get argument position for Simulink model port from model-specific C function prototype
`getArgQualifier (Function Prototype Control)`	Get argument type qualifier for Simulink model port from model-specific C function prototype

`getDefaultConf (Function Prototype Control)`	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
`getFunctionName`	Get function name from model-specific C function prototype
`getNumArgs (Function Prototype Control)`	Get number of function arguments from model-specific C function prototype
`getPreview`	Get model-specific C function prototype code preview
`RTW.configSubsystemBuild`	Open GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
`RTW.getFunctionSpecification`	Get handle to a model-specific C prototype function control object
`runValidation (Function Prototype Control)`	Validate model-specific C function prototype against Simulink model to which it is attached
`setArgCategory (Function Prototype Control)`	Set argument category for Simulink model port in model-specific C function prototype
`setArgName (Function Prototype Control)`	Set argument name for Simulink model port in model-specific C function prototype
`setArgPosition (Function Prototype Control)`	Set argument position for Simulink model port in model-specific C function prototype
`setArgQualifier (Function Prototype Control)`	Set argument type qualifier for Simulink model port in model-specific C function prototype
`setFunctionName`	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

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
`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
`registerCPromotableMacroEntry`	Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)
`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

Processor-in-the-Loop

Connectivity Configuration (p. 1-12)	Define processor-in-the-loop (PIL) configuration
Build (p. 1-12)	Configure PIL build process
Execution Download, Start and and Stop (p. 1-13)	Control downloading, starting and resetting PIL executable on target hardware
Host and Target Communications (p. 1-13)	Configure host-target communications
Host-Side Communications (p. 1-13)	Configure host-side communications channel and drivers
Target-Side Communications (p. 1-13)	Configure target-side communications channel and drivers

Connectivity Configuration

`rtw.connectivity.ComponentArgs`	Provide parameters to each of the target connectivity components
`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 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

2

Functions — Alphabetical List

addAdditionalHeaderFile

Purpose

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

Syntax

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

Arguments

hEntry

Handle to a TFL table entry previously returned by *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

"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

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

Arguments

hEntry

Handle to a TFL table entry previously returned by *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

"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

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

Arguments

hEntry
: Handle to a TFL table entry previously returned by *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');
addAdditionalLinkObjsPath(op_entry, fullfile(libdir, 'bin'));
```

See Also

`addAdditionalLinkObjPath`

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

"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

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

Arguments

hEntry

Handle to a TFL table entry previously returned by *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');
addAdditionalLinkObjsPath(op_entry, fullfile(libdir, 'bin'));
```

See Also

`addAdditionalLinkObj`

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

"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

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

Arguments

hEntry
: Handle to a TFL table entry previously returned by *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

"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

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

Arguments

hEntry
: Handle to a TFL table entry previously returned by *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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addArgConf

Purpose

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

Syntax

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

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'`.

Description

The `addArgConf` function 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 will determine the argument position for the port in the function prototype, unless it is changed by other means.

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

Example

In the following example, the `addArgConf` function is used to add argument configuration information for ports `Input` and `Output` in an ERT-based version of `rtwdemo_counter`. After executing these commands, you can click the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box to bring up 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)
```

See Also

`attachToModel (Function Prototype Control)`

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

addConceptualArg

Purpose

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

Syntax

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

Arguments

hEntry

Handle to a TFL table entry previously returned by *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = `RTW.TflCOperationEntry`.

arg

Argument of type `TflArg`, such as returned by `TflArg* 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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addEntry

Purpose

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

Syntax

```
TflEntry 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`

Returns

TFL table entry of type `TflEntry`.

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();

hLib.addEntry( op_entry );
```

See Also

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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

addIOConf

Purpose

Add AUTOSAR I/O configuration to a model

Syntax

```
autosarInterfaceObj.addIOConf(portName, dataAccessMode, autosarPort, interfaceName, dataElement)
```

Description

`addIOConf` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.addIOConf(portName, dataAccessMode, autosarPort, interfaceName, dataElement)` adds an AUTOSAR I/O configuration, corresponding to the given port name, to `autosarInterfaceObj`, the model-specific `RTW.AutosarInterface` object.

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

`portName` is the inport/outport name (string).

`dataAccessMode` is the data access mode, either `'ImplicitSend'`, `'ImplicitReceive'`, or `'ExplicitSend'`, `'ExplicitReceive'` (string).

`autosarPort` is the Interface name (string).

`interfaceName` is the Interface name (string).

`dataElement` is the Interface name (string).

Purpose

Attach `RTW.AutosarInterface` object to model

Syntax

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

Description

`attachToModel` is a method of the class `RTW.AutosarInterface`.

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

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

attachToModel (C++ Encapsulation Interface Control)

Purpose

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

Syntax

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

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.

Description

The `attachToModel` function attaches a model-specific C++ encapsulation interface to a loaded ERT-based Simulink model.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

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

Syntax

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

Arguments

obj

Handle to a model-specific C prototype function control object, such as a handle 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.

Description

The `attachToModel` function attaches a model-specific C function prototype to a loaded ERT-based Simulink model.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

copyConceptualArgsToImplementation

Purpose

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

Syntax

```
void copyConceptualArgsToImplementation(hEntry)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by *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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

createAndAddConceptualArg

Purpose

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

Syntax

```
void createAndAddConceptualArg(hEntry, argType, varargin)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by *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 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.

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

"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

```
void createAndAddImplementationArg(hEntry,
argType, varargin)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by *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`.

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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

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

Syntax

```
void createAndSetCImplementationReturn(hEntry,
argType, varargin)
```

Arguments

hEntry
: Handle to a TFL table entry previously returned by *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`.

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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

createCalibrationComponentObjects

Purpose

Create Simulink calibration objects from AUTOSAR calibration component

Syntax

```
importerObj.createCalibrationComponentObjects(componentName)
[success] = createCalibrationComponentObjects(importerObj. com
   ponentName, CreateSimulinkObject, 'true')
[success] = createCalibrationComponentObjects(importerObj,
   componentName, Property1, Value1)
```

Description

`createCalibrationComponentObjects` is a method of the class `arxml.importer`.

`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.

`componentName` is the absolute short name path of the calibration parameter component.

`[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 the `arxml.importer` object `importerObj`.

`[success] = createCalibrationComponentObjects(importerObj, componentName, Property1, Value1)` can specify the following parameter/value pair:

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

Outputs:

`success`
: True if this function is successful. Otherwise, it is false.

Example

```
importer_obj.createCalibrationComponentObjects('/package/autosar_co
```

See Also

createComponentAsSubsystem;createComponentAsModel; getCalibrationComponentNames; getDependencies; setDependencies; setFile

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

`createComponentAsModel` is a method of the class `arxml.importer` that creates an AUTOSAR atomic software component as a Simulink model.

`[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`. `ComponentName` is the absolute Short-Name path of the atomic software component .

`[modelH, success] = importerObj.createComponentAsModel(ComponentName, Property1, Value1, Property2, Value2...)` creates the Simulink model and can specify the following parameter/value pairs:

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

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

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

Outputs:

`modelH`
Model handle.

`success`
True if this function is successful. Otherwise, it is false.

Example

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

See Also

`createComponentAsSubsystem`;`getComponentNames`; `getDependencies`; `setDependencies`; `setFile`

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

`createComponentAsSubsystem` is a method of the class `arxml.importer` that creates an AUTOSAR atomic software component as a Simulink atomic subsystem.

`[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`. `ComponentName` is the absolute short name path of the atomic software component .

`[susbsysH, success] = importerObj.createComponentAsSubsystem(ComponentName, Property1, Value1, Property2, Value2...)` creates the Simulink subsystem and can specify the following parameter/value pairs:

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

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

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

Outputs:

`susbsysH`
: Subsystem handle.

`success`
: True if this function is successful. Otherwise, it is false.

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**.

Example

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

See Also

`createComponentAsModel`; `getComponentNames`; `getDependencies`; `getFile`; `setDependencies`; `setFile`

getArgCategory (C++ Encapsulation Interface Control)

Purpose

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

Syntax

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

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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

Description

The `getArgCategory` function 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.

See Also

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

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

"Generating and Controlling C++ Encapsulation 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)
```

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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

Description

The `getArgCategory` function 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.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getArgName (C++ Encapsulation Interface Control)

Purpose

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

Syntax

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

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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

Description

The `getArgName` function gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

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

Syntax

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

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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

Description

The `getArgName` function gets the argument name corresponding to a specified Simulink model inport or outport from a specified model-specific C function prototype.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getArgPosition (C++ Encapsulation Interface Control)

Purpose

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

Syntax

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

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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.

Description

The `getArgPosition` function 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.

See Also

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

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

"Generating and Controlling C++ Encapsulation 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)
```

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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.

Description

The `getArgPosition` function 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.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getArgQualifier (C++ Encapsulation Interface Control)

Purpose

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

Syntax

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

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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

Description

The `getArgQualifier` function 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.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

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

Syntax

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

Arguments

obj

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

portName

String specifying the name of an inport or outport in your Simulink model.

Returns

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

Description

The `getArgQualifier` function 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.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getCalibrationComponentNames

Purpose

Get calibration component names

Syntax

```
calibrationComponentNames = importerObj.getCalibrationComponentNames
```

Description

`getCalibrationComponentNames` is a method of the class `arxml.importer`.

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

`componentNames` is a cell array of strings in which each element is the absolute short name path of the corresponding calibration parameters component :

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

See Also

`createCalibrationComponentObjects`;

Purpose

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

Syntax

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

Arguments

obj

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

Returns

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

Description

The `getClassName` function gets the name of the class described by the specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

getComponentName

Purpose

Get XML component name

Syntax

```
componentName = autosarInterfaceObj.getComponentName
```

Description

`getComponentName` is a method of the class `RTW.AutosarInterface`.

`componentName = autosarInterfaceObj.getComponentName` gets the XML component name of the `RTW.AutosarInterface` object specified by `autosarInterfaceObj`.

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

`componentName` is the name of the XML component specified by `autosarInterfaceObj`.

See Also

`setComponentName`

Purpose

Get atomic software component names

Syntax

```
componentNames = importerObj.getComponentNames
```

Description

`getComponentNames` is a method of the class `arxml.importer`.

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

`componentNames` is a 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'
```

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.

See Also

`createComponentAsModel`; `createComponentAsSubsystem`

getDataTypePackageName

Purpose

Get XML data type package name

Syntax

```
dataTypePackageName = autosarInterfaceObj.getDataTypePackageName
```

Description

`getDataTypePackageName` is a method of the class `RTW.AutosarInterface`.

`dataTypePackageName = getDataTypePackageName(autosarInterfaceObj)` gets the XML data type package name of the specified `RTW.AutosarInterface` object, `autosarInterfaceObj`.

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

`dataTypePackageName` is the name of the data type package specified by `autosarInterfaceObj`.

Purpose

Get default configuration

Syntax

```
autosarInterfaceObj.getDefaultConf
```

Description

`getDefaultConf` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.getDefaultConf` gets the model's default configuration for this `RTW.AutosarInterface` object, `autosarInterfaceObj`, using the information from the model to which the object has already been 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.

getDefaultConf (C++ Encapsulation Interface Control)

Purpose

Get default configuration information for model-specific C++ encapsulation interface from Simulink model to which it is attached

Syntax

```
getDefaultConf(obj)
```

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`.

Description

The `getDefaultConf` function 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 (C++ Encapsulation Interface Control)`, to attach the C++ encapsulation interface to a loaded model.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get default configuration information for model-specific C function prototype from Simulink model to which it is attached

Syntax

```
getDefaultConf(obj)
```

Arguments

obj

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

Description

When you initially invoke the `getDefaultConf` function, the specified model-specific C function prototype initializes 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 (Function Prototype Control)`, to attach the function prototype to a loaded model.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getDependencies

Purpose

Get list of XML dependency files

Syntax

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

Description

`getDependencies` is a method of the class `arxml.importer`.

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

`Dependencies` is a cell array of strings.

See Also

`getFile`; `setDependencies`; `setFile`

Purpose

Get runnable execution period

Syntax

```
EP = autosarInterfaceObj.getExecutionPeriod
```

Description

`getExecutionPeriod` is a method of the class `RTW.AutosarInterface`.

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

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

getFile

Purpose

Return XML file name for `arxml.importer` object

Syntax

```
filename = importerObj.getFile
```

Description

`getFile` is a method of the class `arxml.importer`.

`filename = importerObj.getFile` returns the XML `filename` associated with the `arxml.importer` object, `importerObj`.

See Also

`getDependencies`; `setDependencies`; `setFile`

Purpose

Get function name from model-specific C function prototype

Syntax

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

Arguments

obj

Handle to a model-specific C prototype function control object, such as a handle 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.

Returns

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

Description

The `getFunctionName` function gets the name of the step or initialize function described by the specified model-specific C function prototype.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getInitEventName

Purpose

Get initial event name

Syntax

```
initEventName = autosarInterfaceObj.getInitEventName
```

Description

`getInitEventName` is a method of the class `RTW.AutosarInterface`.

`initEventName = autosarInterfaceObj.getInitEventName` gets the initial event name of the `RTW.AutosarInterface` object specified by `autosarInterfaceObj`.

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

`initEventName` is the name of the initial event specified by `autosarInterfaceObj`.

See Also

`setInitEventName`

Purpose

Get initial runnable name

Syntax

```
initRunnableName = autosarInterfaceObj.getInitRunnableName
```

Description

`attachToModel` is a method of the class `RTW.AutosarInterface`.

`initRunnableName = autosarInterfaceObj.getInitRunnableName` gets the initial runnable name of the `RTW.AutosarInterface` object.

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

`initRunnableName` is the name of the initial runnable specified by `autosarInterfaceObj`.

See Also

`setInitRunnableName`

getInterfacePackageName

Purpose

Get XML interface package name

Syntax

```
interfacePkgName = autosarInterfaceObj.getInterfacePackageName
```

Description

`getInterfacePackageName` is a method of the class `RTW.AutosarInterface`.

`interfacePkgName = autosarInterfaceObj.getInterfacePackageName` gets the XML interface package name of the `RTW.AutosarInterface` object specified by `autosarInterfaceObj`.

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

`interfacePkgName` is the name of the interface package specified by `autosarInterfaceObj`.

Purpose

Get XML internal behavior name

Syntax

```
internalBehaviorName = autosarInterfaceObj.getInternalBehaviorName
```

Description

`getInternalBehaviorName` is a method of the class `RTW.AutosarInterface`.

`internalBehaviorName = autosarInterfaceObj.getInternalBehaviorName` gets the XML internal behavior name of the `RTW.AutosarInterface` object specified by `autosarInterfaceObj`.

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

`internalBehaviorName` is the name of the internal behavior specified by `autosarInterfaceObj`.

getImplementationName

Purpose

Get XML implementation name

Syntax

```
implementationName = autosarInterfaceObj.getImplementationName
```

Description

`getImplementationName` is a method of the class `RTW.AutosarInterface`.

`implementationName = autosarInterfaceObj.getImplementationName` gets the XML implementation name of the `RTW.AutosarInterface` object specified by `autosarInterfaceObj`.

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

`implementationName` is the name of the implementation specified by `autosarInterfaceObj`.

Purpose

Get I/O AUTOSAR port name

Syntax

```
ioAutosarName = autosarInterfaceObj.getIOAutosarPortName(portName)
```

Description

`getIOAutosarPortName` is a method of the class `RTW.AutosarInterface`.

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

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

`portName` is the inport/outport name (string).

`ioAutosarName` is the AUTOSAR port name of the given `portName`.

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

See Also

`setIOAutosarPortName`

getIODataAccessMode

Purpose

Get I/O data access mode

Syntax

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

Description

`getIODataAccessMode` is a method of the class `RTW.AutosarInterface`.

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

`portName` is the inport/outport name (string).

`dataAccessMode` returns the data access mode of the given port, one of `'ImplicitSend'`, `'ImplicitReceive'`, or `'ExplicitSend'`, `'ExplicitReceive'` (string).

See Also

`setIODataAccessMode`

Purpose

Get I/O data element name

Syntax

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

Description

`getIODataElement` is a method of the class `RTW.AutosarInterface`.

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

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

`portName` is the inport/outport name (string).

`ioDataElement` is the data element of the given `portName` (string).

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

See Also

`setIODataElement`

getIOErrorStatusReceiver

Purpose

Get receiver port name

Syntax

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

Description

`getIOErrorStatusReceiver` is a method of the class `RTW.AutosarInterface`.

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

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

`PortName` is the inport/outport name (string).

`ESR` is the receiver port name of the given `PortName`.

Purpose

Get I/O interface name

Syntax

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

Description

`getIOInterfaceName` is a method of the class `RTW.AutosarInterface`.

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

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

`portName` is the inport/outport name (string).

`ioInterfaceName` is the I/O interface name of the given `portName`.

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

See Also

`setIOInterfaceName`

getIOPortNumber

Purpose

Get I/O AUTOSAR port name

Syntax

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

Description

`getIOPortNumber` is a method of the class `RTW.AutosarInterface`.

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

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

`PortName` is the inport/outport name (string).

`IOPortNumber` is the port number of the given `PortName`.

Purpose

Get port I/O service interface

Syntax

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

Description

`getIOServiceInterface` is a method of the class `RTW.AutosarInterface`.

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

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

`PortName` is the inport/outport name (string).

`SI` is the I/O service interface of the given `PortName`.

getIOServiceName

Purpose

Get port I/O service name

Syntax

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

Description

`getIOServiceName` is a method of the class `RTW.AutosarInterface`.

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

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

`PortName` is the inport/outport name (string).

`SN` is the I/O service name of the given `PortName`.

Purpose

Get port I/O service operation

Syntax

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

Description

`getIOServiceOperation` is a method of the class `RTW.AutosarInterface`.

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

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

`PortName` is the inport/outport name (string).

`SO` is the I/O service operation of the given `PortName`.

getNumArgs (C++ Encapsulation Interface Control)

Purpose

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

Syntax

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

Arguments

obj

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

Returns

An integer specifying the number of step method arguments.

Description

The `getNumArgs` function gets the number of arguments for the step method described by the specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation 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)
```

Arguments

obj

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

Returns

An integer specifying the number of function arguments.

Description

The `getNumArgs` function gets the number of function arguments for the function described by the specified model-specific C function prototype.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getPeriodicEventName

Purpose

Get periodic event name

Syntax

```
periodicEventName = autosarInterfaceObj.getPeriodicEventName
```

Description

`getPeriodicEventName` is a method of the class `RTW.AutosarInterface`.

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

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

`periodicEventName` is the name of the periodic event specified by `autosarInterfaceObj`.

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

`setPeriodicEventName`

Purpose

Get periodic runnable name

Syntax

```
periodicRunnableName = autosarInterfaceObj.getPeriodicRunnableName
```

Description

`getPeriodicRunnableName` is a method of the class `RTW.AutosarInterface`.

`periodicRunnableName = autosarInterfaceObj.getPeriodicRunnableName` gets the name of the periodic runnable specified in `autosarInterfaceObj`, an `RTW.AutosarInterface` object.

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

`periodicRunnableName` is the name of the periodic runnable specified by `autosarInterfaceObj`.

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

`setPeriodicRunnableName`

getPortDefaultConf

Purpose

Get port default configuration

Syntax

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

Description

`getPortDefaultConf` is a method of the class `RTW.AutosarInterface`.

`[autosarPort, interfaceName, dataElement, mode]=autosarInterfaceObj.getPortDefaultConf(portH)` gets the port's default configuration for `autosarInterfaceObj`, the `RTW.AutosarInterface` object.

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

`autosarPort` is the AUTOSAR port name of the given `portH` (string).

`interfaceName` is the I/O interface name of the given `portH` (string).

`dataElement` is the data element of the given `portH` (string).

`dataAccessMode` returns the data access mode of the given port, one of `'ImplicitSend'`, `'ImplicitReceive'`, or `'ExplicitSend'`, `'ExplicitReceive'` (string).

Purpose

Get model-specific C function prototype code preview

Syntax

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

Arguments

obj

Handle to a model-specific C prototype function control object, such as a handle 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.

Returns

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

Description

The `getPreview` function gets the model-specific C function prototype code preview.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

getStepMethodName

Purpose

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

Syntax

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

Arguments

obj

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

Returns

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

Description

The `getStepMethodName` function gets the name of the step method described by the specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

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

Syntax

```
TflArg* 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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Construct `arxml.importer` object

Syntax

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

Description

`importer` is a method of the class `arxml.importer`.

`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.

See Also

`createComponentAsModel`; `createComponentAsSubsystem`; `getDependencies`; `getFile`; `setDependencies`; `setFile`

model_initialize

Purpose

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

Syntax

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

Arguments

firstTime

The Real-Time Workshop Embedded Coder software generates this argument for Simulink models only if the `IncludeERTFirstTime` model configuration parameter is set to `on`. Use of the *`firstTime`* argument will be discontinued in a future release (see the note below).

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.

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 the `IncludeERTFirstTime` model configuration parameter is set to `on`, the generated code passes in *`firstTime`* as `1 (TRUE)`.

Note In a future release, the Real-Time Workshop Embedded Coder software will no longer use the *`firstTime`* argument in a model's generated *`model`*`_initialize` function. For more information about the `IncludeERTFirstTime` model configuration parameter and a related target configuration parameter, `ERTFirstTimeCompliant`, see "Parameter Command-Line Information Summary" in the Real-Time Workshop documentation.

See Also

`model_SetEventsForThisBaseStep`, `model_step`, `model_terminate`

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

model_SetEventsForThisBaseStep

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 the 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

model_step

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 Model Interfaces" and "Controlling Model 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" 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

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

Purpose

Create TFL function entry based on specified parameters and register in TFL table

Syntax

```
TflCFunctionEntry* 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:

Floating-Point Math Functions			
abs	cos	log10	tan
acos	cosh	pow (Simulink)/power (Embedded MATLAB™)	tanh
asin	exp	sin	
atan	floor	sinh	

`ceil`	`log`	`sqrt`	
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.

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`, ..., `u`*`n`*
  - 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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)

Syntax

```
TflCFunctionEntry* 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.

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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

RTW.configSubsystemBuild

Purpose

Open GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem

Syntax

```
RTW.configSubsystemBuild(block)
```

Arguments

block

String specifying the name of a nonvirtual subsystem block in an ERT-based Simulink model.

Description

The `RTW.configSubsystemBuild` function opens a graphical user interface that allows you to configure either C function prototype information or C++ encapsulation interface information to be used in right-click builds of the specified subsystem. The appropriate dialog box is opened based on the **Language** value currently 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, the following requirements must be met:

- The system target file `ert.tlc` must be selected for the model.
- The **Language** parameter value `C++ (Encapsulated)` must be selected for the model.
- The subsystem must be convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

See Also

"Configuring Function Prototypes for Nonvirtual Subsystems" in the Real-Time Workshop Embedded Coder documentation

"Controlling Model 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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

RTW.getEncapsulationInterfaceSpecification

Purpose

Get handle to model-specific C++ encapsulation interface control object

Syntax

```
obj = RTW.getEncapsulationInterfaceSpecification(modelName)
```

Arguments

modelName

String specifying the name of a loaded ERT-based Simulink model.

Returns

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 `[]`.

Description

The `RTW.getEncapsulationInterfaceSpecification` function returns a handle to the model-specific C++ encapsulation interface control object.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Get handle to a model-specific C prototype function control object

Syntax

```
obj = RTW.getFunctionSpecification(modelName)
```

Arguments

modelName

String specifying the name of a loaded ERT-based Simulink model.

Returns

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 `[]`.

Description

The `RTW.getFunctionSpecification` function returns a handle to the model-specific C function prototype control object.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

rtiostreamClose

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 Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_rtiostream`

`rtwdemo_custom_pil`

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 non-negative 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 Your 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 Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_rtiostream`

`rtwdemo_custom_pil`

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 Your Target" in the Real-Time Workshop Embedded Coder documentation.

```
rtwdemo_rtiostream
```

```
rtwdemo_custom_pil
```

rtiostream_wrapper

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 M-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 Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_rtiostream`

`rtwdemo_custom_pil`

Purpose

Provide parameters to each of the target connectivity components

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 Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

rtw.connectivity.Config

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 "PIL Feature 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 Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

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 2-125 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 non-makefile 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: 1.1.4.6 $

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 Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

```
rtwdemo_custom_pil
```

rtw.connectivity.Launcher

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.
	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 Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

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 Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

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.

Method	Syntax and Description
`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 used in the `mypil` host example, enter: `obj.addPILMain(`host')`

See Also

`rtw.connectivity.ComponentArgs`, `rtiostream_wrapper`

"Verifying Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

"Build Information Object" in the Real-Time Workshop documentation.

```
rtwdemo_custom_pil
```

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. With TCP/IP, you need only supply the target-side drivers.

For other communications protocols (e.g., serial, 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 sub-class of `rtw.connectivity.RtIOStreamHostCommunicator`. This may be necessary when more complex configuration is required. For example, the demo sub-class `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 verbose or silent operation.

See Also

`rtw.connectivity.ComponentArgs`, `rtw.connectivity.Launcher`, `rtiostream_wrapper`

"Verifying Generated Code with Processor-in-the-Loop" in the Real-Time Workshop Embedded Coder documentation.

"Creating a Connectivity Configuration for Your Target" in the Real-Time Workshop Embedded Coder documentation.

`rtwdemo_custom_pil`

Purpose

Validate `RTW.AutosarInterface` object against model

Syntax

```
[status, msg] = autosarInterfaceObj.runValidation
```

Description

`runValidation` is a method of the class `RTW.AutosarInterface`.

`[Status, Message] = autosarInterfaceObj.runValidation` runs a validation check for the `RTW.AutosarInterface` object against the model to which the object is attached. Before calling this method, you must call the `attachToModel` method.

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

`Status` returns a status flag indicating whether the configuration is valid. If valid, `status` is true; otherwise, it is false.

`Message`: If the returned status flag is false, `Message` stores the explanation of why the configuration is invalid.

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.

Validation Checks (Continued)

Group	Check
Unsupported features	Model must not contain custom code blocks.
	Model must not contain continuous time.
	Model must not contain non-inlined S-functions.
	Model must not contain non-finite 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.
	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.

Definitions of 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.

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.)
	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.

Multiple Runnable Validation Checks (Continued)

Group	Check
Other multiple runnable validation checks	All runnable names, event names, and interrunnable variable names must be unique. Lines representing interrunnable variables must be labeled 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, non-complex types. This is required by the AUTOSAR specification. Signal lines that connect two top-level function-call subsystems represent interrunnable variables.
	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.

runValidation (C++ Encapsulation Interface Control)

Purpose

Validate model-specific C++ encapsulation interface against Simulink model to which it is attached

Syntax

```
[status, msg] = runValidation(obj)
```

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.getClassInterfaceSpecification(modelName)`.

Returns

`[status, msg]`, where *status* is true for a valid configuration and false otherwise. If *status* is false, *msg* contains a string of information explaining why the configuration is invalid.

Description

The `runValidation` function 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 (C++ Encapsulation Interface Control)`, to attach a function prototype to a loaded model, or `RTW.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

Purpose

Validate model-specific C function prototype against Simulink model to which it is attached

Syntax

```
[status, msg] = runValidation(obj)
```

Arguments

obj

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

Returns

`[status, msg]`, where *status* is true for a valid configuration and false otherwise. If *status* is false, *msg* contains a string of information explaining why the configuration is invalid.

Description

The `runValidation` function 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 (Function Prototype Control)`, 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.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

setArgCategory (C++ Encapsulation Interface Control)

Purpose

Set argument category for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgCategory(obj, portName, category)
```

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.getClassInterfaceSpecification(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 will cause the argument to move to the first argument position when `attachToModel (C++ Encapsulation Interface Control)` or `runValidation (C++ Encapsulation Interface Control)` is called.

Description

The `setArgCategory` function 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.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

setArgCategory (Function Prototype Control)

Purpose

Set argument category for Simulink model port in model-specific C function prototype

Syntax

```
setArgCategory(obj, portName, category)
```

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'`, 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 will cause the argument to move to the first argument position when `attachToModel (Function Prototype Control)` or `runValidation (Function Prototype Control)` is called.

Description

The `setArgCategory` function 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.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument name for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgName(obj, portName, argName)
```

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.getClassInterfaceSpecification(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.

Description

The `setArgName` function sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

setArgName (Function Prototype Control)

Purpose

Set argument name for Simulink model port in model-specific C function prototype

Syntax

```
setArgName(obj, portName, argName)
```

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.

Description

The `setArgName` function sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument position for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgPosition(obj, portName, position)
```

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.getClassInterfaceSpecification(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.

Description

The `setArgPosition` function 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++ encapsulation interface. The specified argument will be moved to the specified position, and other arguments will be shifted by one position accordingly.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

setArgPosition (Function Prototype Control)

Purpose

Set argument position for Simulink model port in model-specific C function prototype

Syntax

```
setArgPosition(obj, portName, position)
```

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.

Description

The `setArgPosition` function 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 will be moved to the specified position, and other arguments will be shifted by one position accordingly.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface

Syntax

```
setArgQualifier(obj, portName, qualifier)
```

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.getClassInterfaceSpecification(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.

Description

The `setArgQualifier` function sets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

setArgQualifier (Function Prototype Control)

Purpose

Set argument type qualifier for Simulink model port in model-specific C function prototype

Syntax

```
setArgQualifier(obj, portName, qualifier)
```

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.

Description

The `setArgQualifier` function 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.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set class name in model-specific C++ encapsulation interface

Syntax

```
setClassName(obj, clsName)
```

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.getClassInterfaceSpecification(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.

Description

The `setClassName` function sets the class name in the specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

setComponentName

Purpose

Set XML component name

Syntax

```
componentName = autosarInterfaceObj.setComponentName
```

Description

`setComponentName` is a method of the class `RTW.AutosarInterface`.

`componentName = autosarInterfaceObj.setComponentName` sets the XML component name of the `RTW.AutosarInterface` object specified by `autosarInterfaceObj`.

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

`componentName` is the name of the XML component to be set to `autosarInterfaceObj`

Purpose

Set XML file dependencies

Syntax

```
importerObj.setDependencies(Dependencies)
```

Description

`setDependencies` is a method of the class `arxml.importer`.

`importerObj.setDependencies(Dependencies)` sets the XML file dependencies `Dependencies` associated with the `arxml.importer` object, `importerObj`.

`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

`getDependencies`; `getFile`; `setFile`

setFile

Purpose

Set XML file name for `arxml.importer` object

Syntax

```
importerObj.setFile(filename)
```

Description

`setFile` is a method of the class `arxml.importer`.

`filename = importerObj.setFile` sets the XML `filename` associated with the `arxml.importer` object, `importerObj`.

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

See Also

`getDependencies`; `getFile`; `setDependencies`

Purpose

Set function name in model-specific C function prototype

Syntax

```
setFunctionName(obj, fcnName, fcnType)
```

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.

Description

The `setFunctionName` function sets the step or initialization function name in the specified function control object.

See Also

"Controlling Model Function Prototypes" in the Real-Time Workshop Embedded Coder documentation

setInitEventName

Purpose

Set initial event name

Syntax

```
autosarInterfaceObj.setInitEventName(initEventName)
```

Description

`setInitEventName` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setInitEventName(initEventName)` sets the initial event name to the specified `AutosarInterface` object.

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

`initEventName` is the name of the initial event to be set to `autosarInterfaceObj`.

Purpose

Set initial runnable name

Syntax

```
autosarInterfaceObj.setInitRunnableName(initRunnableName)
```

Description

`setInitRunnableName` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setInitRunnableName(initRunnableName)` sets the initial runnable name for the specified `AutosarInterface` object.

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

`initRunnableName` is the initial runnable name to be set to `autosarInterfaceObj`.

setIOAutosarPortName

Purpose

Set AUTOSAR port name

Syntax

```
autosarInterfaceObj.setIOAutosarPortName(portName,autosarPort)
```

Description

`setIOAutosarPortName` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setIOAutosarPortName(portName,autosarPort)` updates the AUTOSAR port name in the configuration for the specified port.

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

`portName` is the inport/outport name (string).

`autosarPort` is the AUTOSAR port name to be set (string).

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

See Also

`getIOAutosarPortName`

Purpose

Set I/O data access mode

Syntax

```
autosarInterfaceObj.setIODataAccessMode(portName,dataAccessMode)
```

Description

`setIODataAccessMode` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setIODataAccessMode(portName,dataAccessMode)` sets the data access mode in the configuration for the specified port.

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

`portName` is the inport/outport name (string).

`dataAccessMode` is the data access mode to be set, one of `'ImplicitSend'`, `'ImplicitReceive'`, or `'ExplicitSend'`, `'ExplicitReceive'` (string).

setIODataElement

Purpose

Set I/O data element

Syntax

```
autosarInterfaceObj.setIODataElement(portName,dataElement)
```

Description

`setIODataElement` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setIODataElement(portName,dataElement)` updates the Data Element in the configuration for the specified port.

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

`portName` is the inport/outport name (string)

`dataElement` is the data element to be set (string).

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

See Also

`getIODataElement`

Purpose

Set I/O interface name

Syntax

```
autosarInterfaceObj.setIOInterfaceName(portName,interfaceName)
```

Description

`setIOInterfaceName` is a method of the class `RTW.AutosarInterface`.

`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.

`portName` is the inport/outport name (string).

`interfaceName` is the I/O interface name to be set (string).

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

See Also

`getIOInterfaceName`

setPeriodicEventName

Purpose

Set periodic event name

Syntax

```
autosarInterfaceObj.setPeriodicEventName(periodicEventName)
```

Description

`setPeriodicEventName` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setPeriodicEventName(periodicEventName)` sets the periodic event name to the `AutosarInterface` object.

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

`periodicEventName` is the name of the periodic event to be set to `autosarInterfaceObj`.

Purpose

Set periodic runnable name

Syntax

```
autosarInterfaceObj.setPeriodicRunnableName(periodicRunnableName)
```

Description

`setPeriodicRunnableName` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.setPeriodicRunnableName(periodicRunnableName)` sets the periodic runnable name to the `RTW.AutosarInterface` object, `autosarInterfaceObj`.

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

`periodicRunnableName` is the periodic runnable name to be set to `autosarInterfaceObj`.

setReservedIdentifiers

Purpose

Register specified reserved identifiers to be associated with TFL table

Syntax

```
void 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, wile the other three (if present) must be associated with ANSI®[1], ISO®[2], or GNU®[3] libraries.

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'};

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';
```

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}.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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

Purpose

Set step method name in model-specific C++ encapsulation interface

Syntax

```
setStepMethodName(obj, fcnName)
```

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.getClassInterfaceSpecification(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.

Description

The `setStepMethodName` function sets the step method name in the specified model-specific C++ encapsulation interface.

See Also

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

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

"Generating and Controlling C++ Encapsulation Interfaces" in the Real-Time Workshop Embedded Coder documentation

setTflCFunctionEntryParameters

Purpose

Set specified parameters for function entry in TFL table

Syntax

```
void setTflCFunctionEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by *hEntry* = `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:

Floating-Point Math Functions			
`abs`	`cos`	`log10`	`tan`
`acos`	`cosh`	`pow` (Simulink)/`power` (Embedded MATLAB)	`tanh`
`asin`	`exp`	`sin`	
`atan`	`floor`	`sinh`	
`ceil`	`log`	`sqrt`	
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 `''`.

`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 Floating-Point 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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

setTflCOperationEntryParameters

Purpose

Set specified parameters for operator entry in TFL table

Syntax

```
void setTflCOperationEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a TFL table entry previously returned by *hEntry* = `RTW.TflCOperationEntry`.

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 `NetSlopeFixedExponent`, 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:

- `'RTW_OP_ADD'` for + (addition)
- `'RTW_OP_MINUS'` for – (subtraction)

- `'RTW_OP_MUL'` for `*` (multiplication)
- `'RTW_OP_DIV'` for `/` (division)

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 NetSlopeFixedExponent 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.

`NetSlopeFixedExponent`
: 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 `''`.

`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 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

"Target Function Libraries" in the Real-Time Workshop Embedded Coder documentation

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 Embedded Coder documentation

"Parameter Command-Line Information Summary" in the Real-Time Workshop documentation

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 Embedded Coder documentation

"Parameter Command-Line Information Summary" in the Real-Time Workshop documentation

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 Embedded Coder documentation

"Parameter Command-Line Information Summary" in the Real-Time Workshop documentation

Purpose

Synchronize configuration with model

Syntax

```
autosarInterfaceObj.syncWithModel
```

Description

`syncWithModel` is a method of the class `RTW.AutosarInterface`.

`autosarInterfaceObj.syncWithModel` synchronizes the configuration with the model for `RTW.AutosarInterface` class.

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

3

Block Reference

Configuration Wizards

Custom M-file	Automatically update active configuration parameters of parent model using custom M-file
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

4

Blocks — Alphabetical List

Custom M-file

Purpose

Automatically update active configuration parameters of parent model using custom M-file

Library

Configuration Wizards

Description

When you add a Custom M-file block to your Simulink model and double-click it, a custom M-file 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 M-file 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 M-file 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 M-file 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

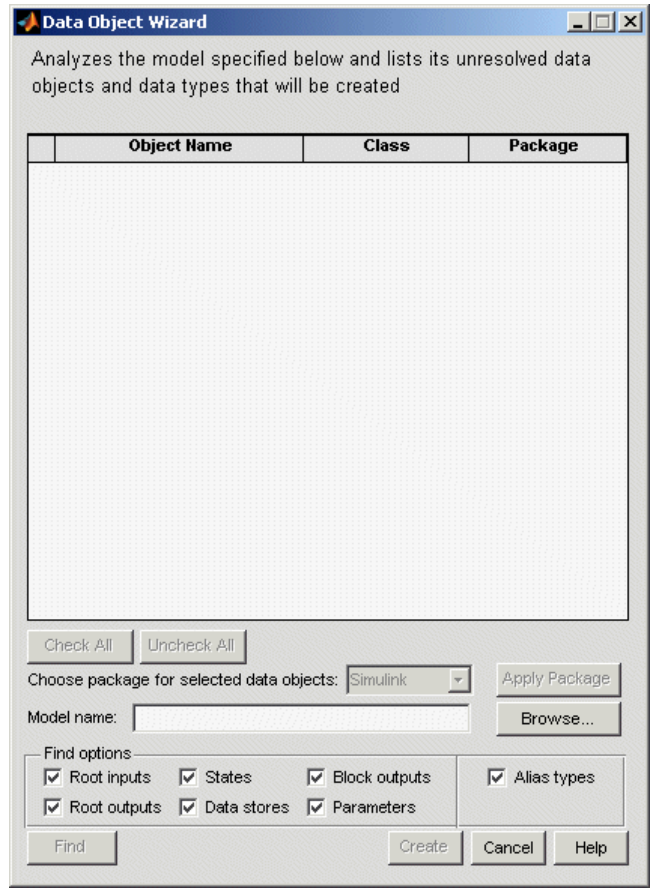
Simulink data object wizard for creating potential Simulink data objects

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

"Creating a Data Dictionary for a Model" 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 M-file 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 M-file 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 M-file 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 M-file 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)

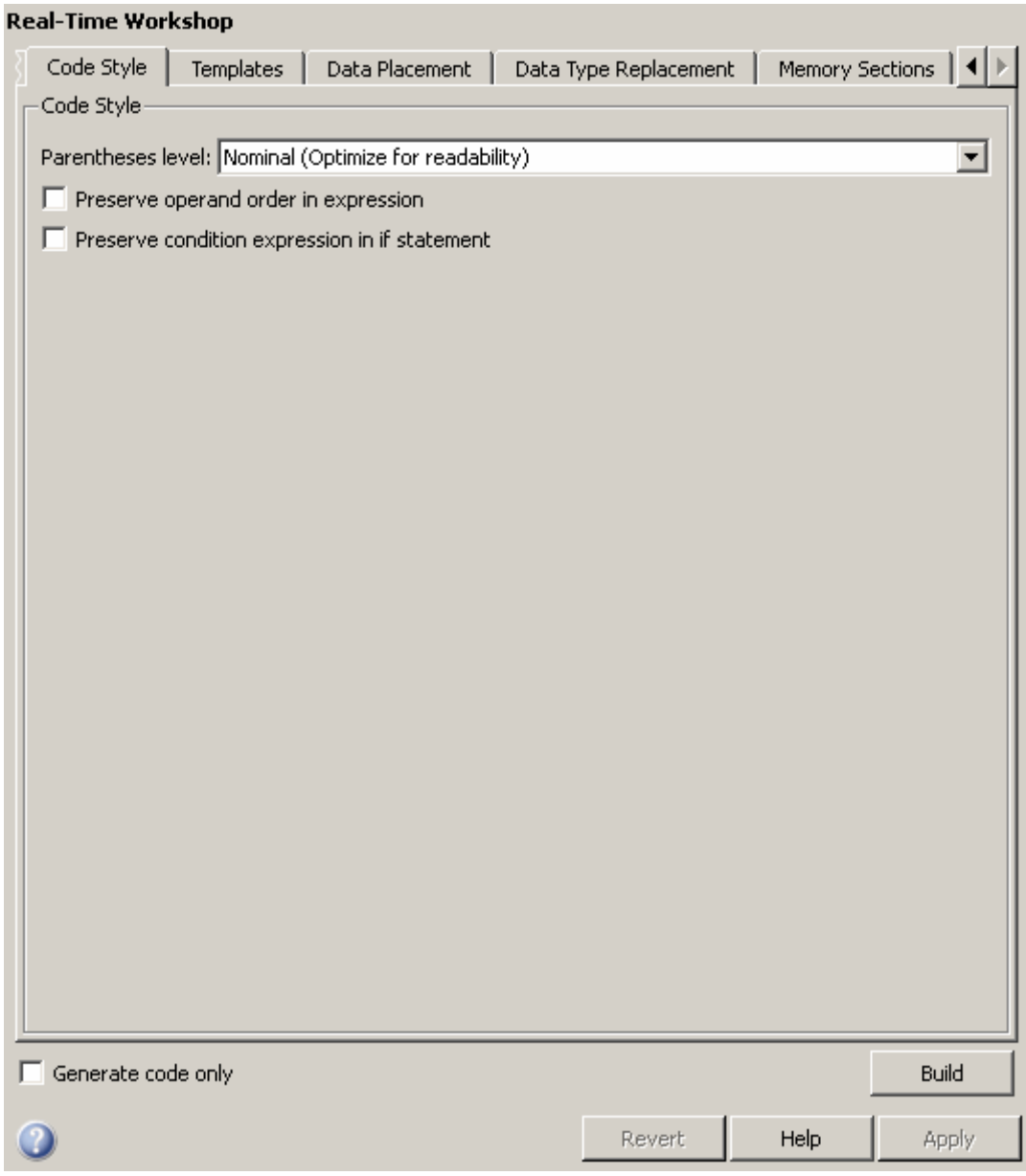
"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop Embedded Coder documentation

GRT (optimized for fixed/floating-point)

5

Configuration Parameters

Real-Time Workshop Pane: Code Style



In this section...

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

Code Style Pane

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
```

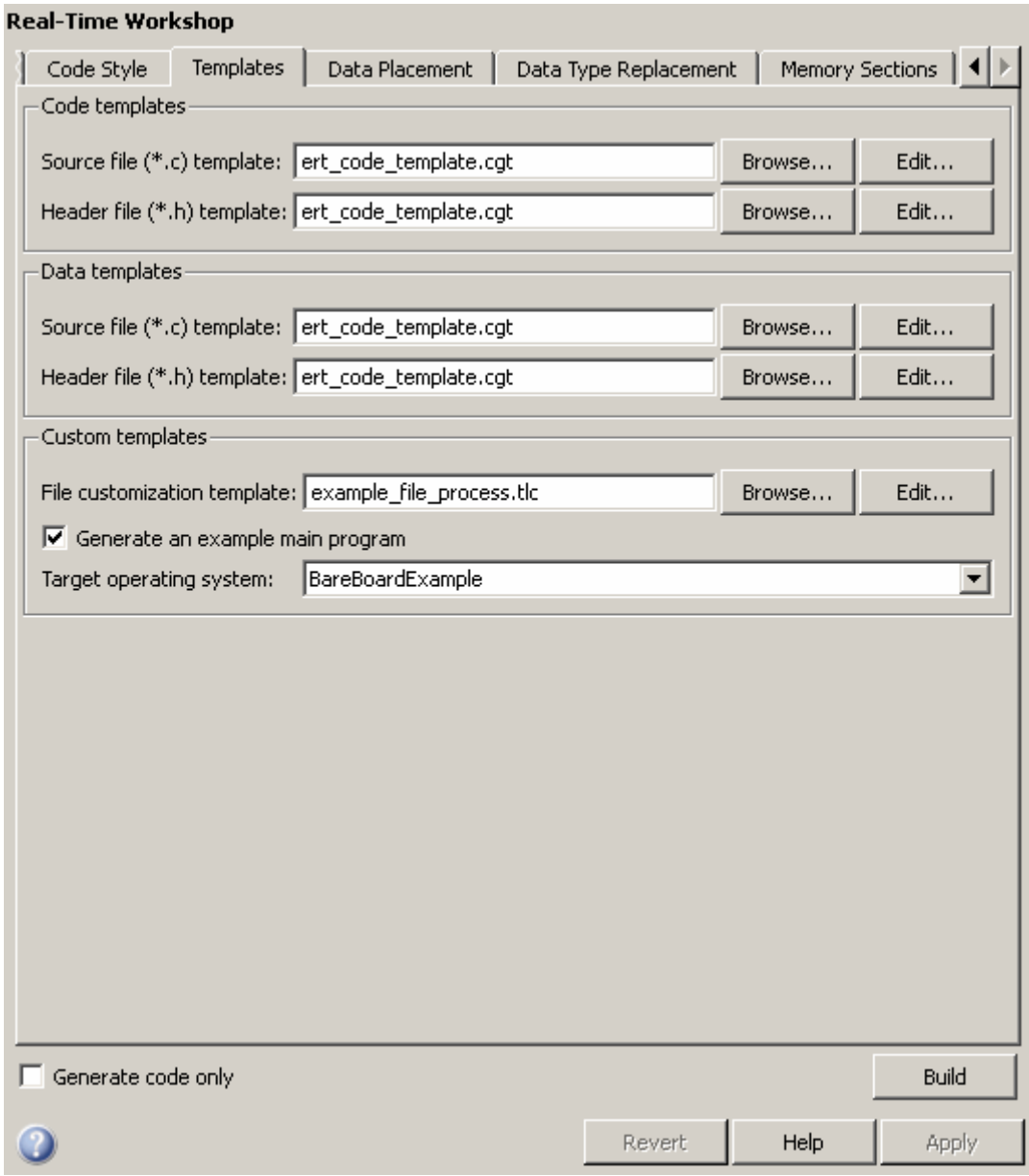
Command-Line Information

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

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	Off

Real-Time Workshop Pane: Templates



In this section...

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

Module Packaging Features

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 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 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 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 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.cgt`

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 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

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 the Main Program Module
- 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'`

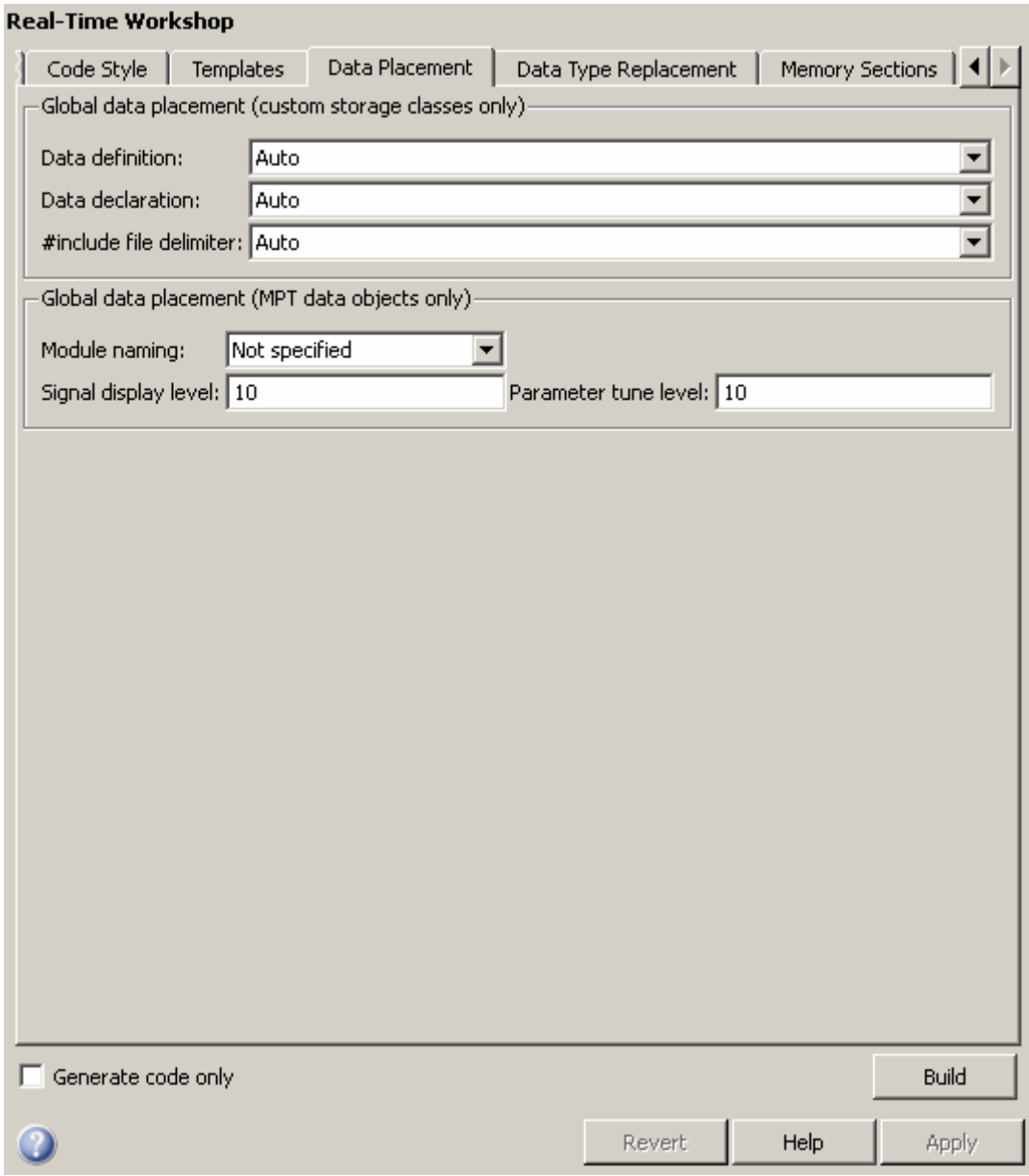
Recommended Settings

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

See Also

- Generating the Main Program Module
- Static Main Program Module
- Custom File Processing

Real-Time Workshop Pane: Data Placement



In this section...

Data 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

Module Packaging Features

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 File 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 File 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: `DataDefinitionFile`
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`

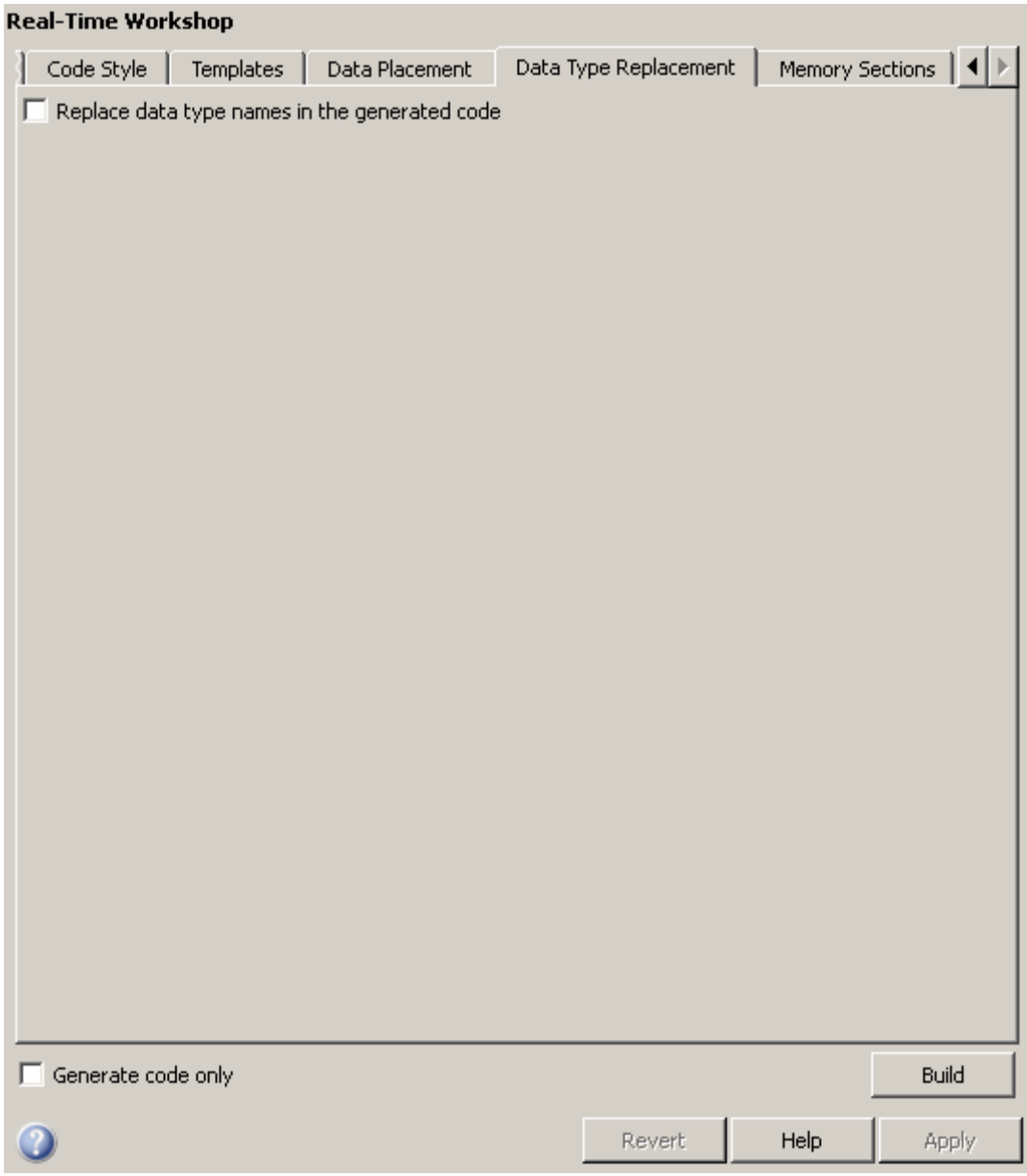
Recommended Settings

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

Real-Time Workshop Pane: Data Type Replacement



In this section...

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. Specify 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

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`, `uint8`, and `boolean`, the `BaseType` of the replacement data type must match the built-in data type.
- For `int`, `uint`, and `char`, the size of the replacement data type must match the size displayed for `int` or `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	Off

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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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 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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
Efficiency	No impact
Safety precaution	''

See Also

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`
Type: string
Value: any valid value
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid value

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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`
Type: string
Value: any valid string
Default: `''`

Recommended Settings

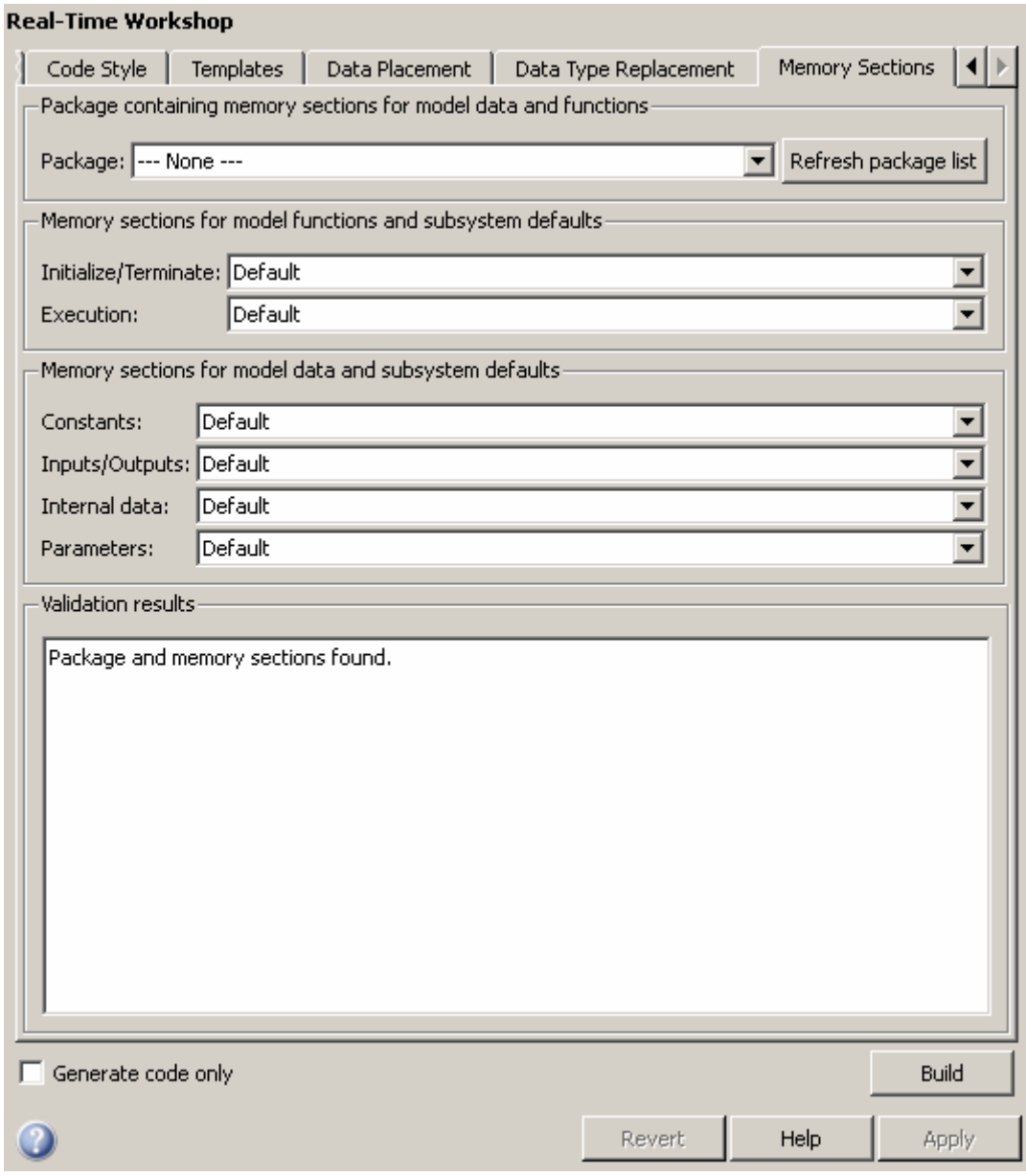
Application	Setting
Debugging	No impact
Traceability	Any valid string

Application	Setting
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...

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

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

Application	Setting
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.

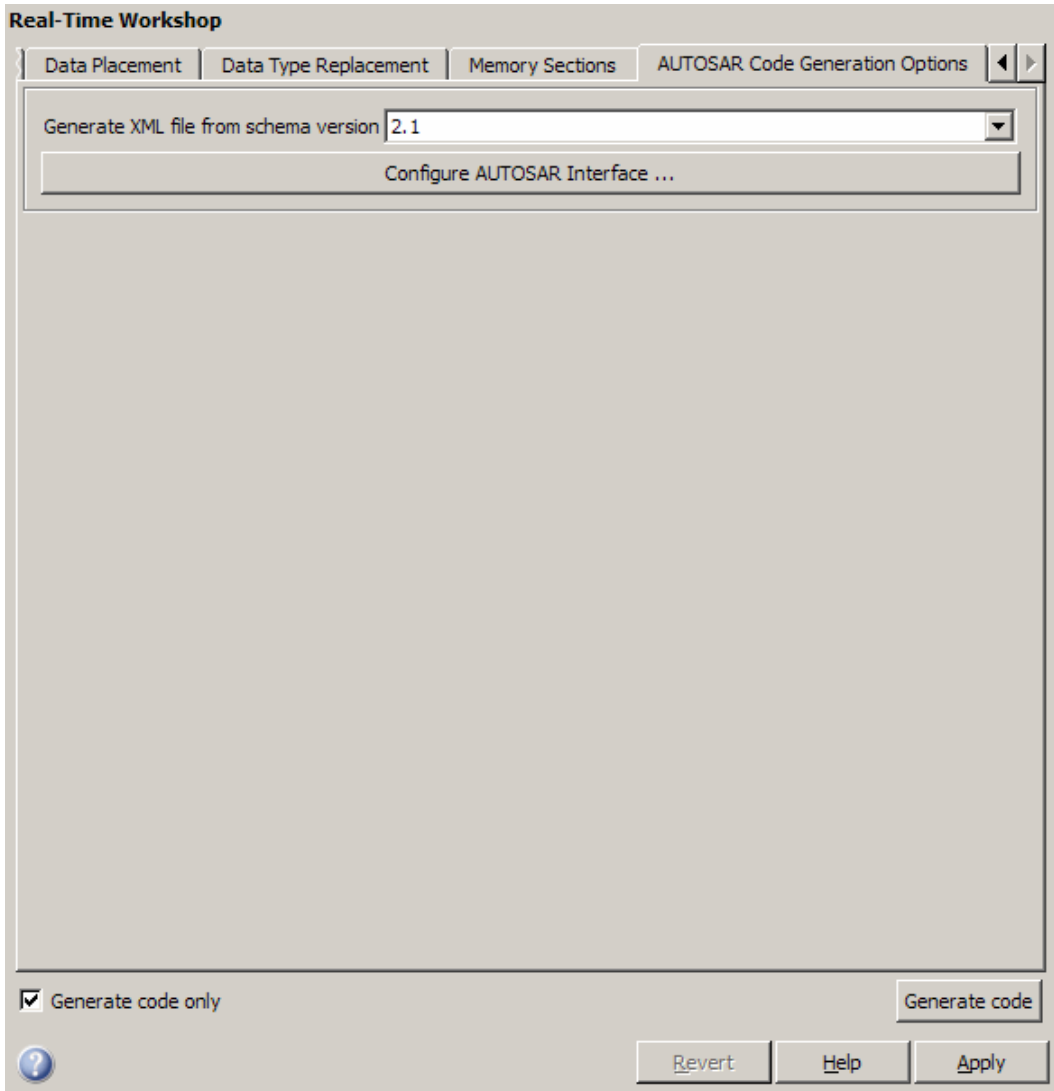
Settings

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...

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 That Complies with AUTOSAR Standards"
- "AUTOSAR Configuration" on page 1-3
- "AUTOSAR" on page 1-2

Generate XML file from schema version

Select the AUTOSAR schema version to use when generating XML files.

Settings

Default: `2.1`

`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: `'2.1' | '2.0'`
Default: `'2.1'`

See Also

"Generating Code That Complies with AUTOSAR Standards"

Configure AUTOSAR Interface

Opens the Model Interface dialog box where you can configure all other AUTOSAR options.

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 That Complies with AUTOSAR Standards"

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	`Non-Hierarchical`
Remove root level I/O zero initialization	No impact	No impact	On	Off	Off
Remove internal data zero initialization	No impact	No impact	On	Off	Off

Mapping of Application Requirements to the Optimization Pane (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Optimize initialization code for model reference	No impact	No impact	On	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

Mapping of Application Requirements to the Real-Time Workshop Pane: Report Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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	''

Mapping of Application Requirements to the Real-Time Workshop Pane: Comments Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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	RN$M	RN$M
Global types	No impact	Any valid combination of tokens	No impact	NR$M	&NRM
Field name of global types	No impact	Any valid combination of tokens	No impact	NM	NM
Subsystem methods	No impact	Any valid combination of tokens	No impact	RNMF	RNMF
Local temporary variables	No impact	Any valid combination of tokens	No impact	NM	NM
Local block output variables	No impact	Any valid combination of tokens	No impact	rtb_NM	rtb_NM

Mapping of Application Requirements to the Real-Time Workshop Pane: Symbols Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Constant macros	No impact	Any valid combination of tokens	No impact	RN$M	RN$M
Minimum mangle length	No impact	1	No impact	4	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`
Signal naming	No impact	`Force uppercase`	No impact	No impact	`None`
M-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 for integer only	No impact	On
Support complex numbers	No impact	No impact	Off for real only	No impact	On
Support non-finite numbers	No impact	No impact	Off	Off	On

Mapping of Application Requirements to the Real-Time Workshop Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Support absolute time	No impact	No impact	Off	Off	On
Support continuous time	No impact	No impact	Off	Off	Off
Support non-inlined S-functions	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	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	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`
"Parameters and states members private"	No impact	No impact	No impact	On	On
"Parameters and states access methods"	On	No impact	No impact	Off	Off
"Generate destructor"	No impact	No impact	No impact	Off	On
"I/O access methods"	On	No impact	No impact	Off	Off
"Inline access methods"	On	On	On	No impact	Off

Mapping of Application Requirements to the Real-Time Workshop Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Suppress error status in real-time model data structure	Off	No impact	On	On	Off
Create Simulink (S-Function) block	On	No impact	No impact	No impact	Off
Enable portable word sizes	On	On	Off	No impact	Off
MAT-file logging	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)`

Mapping of Application Requirements to the Real-Time Workshop Pane: Code Style Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Preserve operand order in expression	On	On	Off	On	Off
Preserve condition expression in if statement	On	On	Off	Off	Off

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`

Mapping of Application Requirements to the Real-Time Workshop Pane: Templates Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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	`BareBoardExample`

Mapping of Application Requirements to the Real-Time Workshop Pane: Data 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

Mapping of Application Requirements to the Real-Time Workshop Pane: Data Placement Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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

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	Off	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`

Mapping of Application Requirements to the Real-Time Workshop Pane: Memory Sections Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory 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”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 Fixed-Point Toolbox™ product are marked with (ERT), (Stateflow®), and (Fixed-Point Toolbox), 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.
`NoFixptDivByZeroProtection` (ERT) (Fixed-Point Toolbox) **`off`**, `on`	**Optimization > Remove code that protects against division arithmetic exceptions**	Suppress generation of code that guards against division by zero for fixed-point data.
`OptimizeModelRefInitCode` (ERT) **`off`**, `on`	**Optimization > Optimize initialization code for model reference**	Suppress generation of initialization code to accommodate the case where this model is referred to by a subsystem that resets its states when enabled. Select this option if the model will never be referred to by such a subsystem. 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.

Command-Line Information: Optimization Pane (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.

Command-Line Information: Real-Time Workshop Pane: Report Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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 M-function or TLC function that adds the custom comment.
`EnableCustomComments` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Custom comments (MPT objects only)**	Add a comment above a signal's or parameter's identifier in the generated file.

Command-Line Information: Real-Time Workshop Pane: Comments Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

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

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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 $N - Name of object
`CustomSymbolStrTmpVar` (ERT) *`string`* - **`$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 M-function to control the naming of symbols with `#define` statements. You can set this parameter only if `DefineNamingRule` is set to `Custom`.

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DefineNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > #define naming**	Specify the rule that changes the spelling of all `#define` names.
`IncDataTypeInIds` (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`.

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`InlinedPrmAccess` (ERT) string - **`Literals`**, `Macros`	**Real-Time Workshop > Symbols > Generate scalar inlined parameters as**	Specify whether inlined parameters are coded as numeric constants or macros. Specify `Macros` for more efficient code.
`MangleLength` (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.

Command-Line Information: Real-Time Workshop Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`GenerateAccessMethods` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Parameters and states access methods**	Generate get/set access methods for non-I/O model structures, including states and parameters, in `C++ (Encapsulated)` model code.
`GenerateDestructor` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Generate destructor**	Generate a destructor for the model class in `C++ (Encapsulated)` model code.
`GenerateErtSFunction` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Create Simulink (S-Function) block**	Wrap the generated code inside an S-Function block. This allows you to validate the generated code in a Simulink model.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateIOAccessMethods` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > I/O access methods**	Generate access methods for root-level I/O signals (if possible) in `C++ (Encapsulated)` model code.
`GeneratePrivateDataMembers` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Parameters and states members private**	Specify whether to generate non-I/O model structures, including states and parameters, as private data members in `C++ (Encapsulated)` model code.
`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.
`InlineAccessMethods` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Inline access methods**	Inline generated access methods in `C++ (Encapsulated)` model code.
`MatFileLogging` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > MAT-file logging**	Generate code that logs data to a MAT-file.

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

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

Command-Line Information: Real-Time Workshop Pane: Interface Tab (Continued)

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

Command-Line Information: Real-Time Workshop Pane: Code Style Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ParenthesesLevel` (ERT) string - `Minimum`, **`Nominal`**, `Maximum`	**Real-Time Workshop > Code Style > Parentheses Level**	Control existence of optional parentheses in generated code.
`PreserveExpressionOrder` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve operand order in expression**	Control reordering of commutable expressions.
`PreserveIfCondition` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Code Style > Preserve condition expression in if statement**	Control preservation of `if` statement conditions.

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.
`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.

Command-Line Information: Real-Time Workshop Pane: Templates Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ERTSrcFileBannerTemplate` (ERT) *string* - **`ert_code_template.cgt`**	**Real-Time Workshop > Templates > Source file (*.c or *.cpp) template**	Specify a template that organizes the generated code `.c` or `.cpp` source files.
`GenerateSampleERTMain` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Templates > Generate an example main program**	Generate an example main program that demonstrates how to deploy the generated code. The program is written to the file `ert_main.c` or `ert_main.cpp`.
`TargetOS` (ERT) string - **`BareBoardExample`**, `VxWorksExample`	**Real-Time Workshop > Templates > Target operating system**	Specify the target operating system for the example main `ert_main.c` or `ert_main.cpp`. `BareBoardExample` is a generic example that assumes no operating system. `VxWorksExample` is tailored to the VxWorks[8] real-time operating system.

Command-Line Information: Real-Time Workshop Pane: Data Placement Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DataDefinitionFile` (ERT) *string* - **`global.c`**	**Real-Time Workshop > Data Placement > Data definition filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data definitions.

8. VxWorks is a registered trademark of Wind River Systems, Inc.

Command-Line Information: Real-Time Workshop Pane: Data Placement Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`DataReferenceFile` (ERT) *`string`* - **`global.h`**	**Real-Time Workshop > Data Placement > Data declaration filename**	Specify the name of a single separate `.c` or `.cpp` file that contains global data references.
`GlobalDataDefinition` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateSourceFile`	**Real-Time Workshop > Data Placement > Data definition**	Select the `.c` or `.cpp` file where variables of global scope are defined.
`GlobalDataReference` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateHeaderFile`	**Real-Time Workshop > Data Placement > Data declaration**	Select the `.h` file where variables of global scope are declared (for example, `extern real_T globalvar;`).
`IncludeFileDelimiter` (ERT) string - **`Auto`**, `UseQuote`, `UseBracket`	**Real-Time Workshop > Data Placement > #include file delimiter**	Specify the delimiter to be used for all data objects that do not have a delimiter specified in the `IncludeFile` property.
`ModuleName` (ERT) *`string`* -	**Real-Time Workshop > Data Placement > Module name**	Specify the name of the module that owns this model.
`ModuleNamingRule` (ERT) string - **`Unspecified`**, `SameAsModel`, `UserSpecified`	**Real-Time Workshop > Data Placement > Module naming**	Specify the rule to be used for naming the module.

Command-Line Information: Real-Time Workshop Pane: Data Placement Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ParamTuneLevel` (ERT) `int` - **`10`**	**Real-Time Workshop > Data Placement > Parameter tune level**	Specify whether the code generator is to declare a parameter data object as tunable global data in the generated code.
`SignalDisplayLevel` (ERT) `int` - **`10`**	**Real-Time Workshop > Data Placement > Signal display level**	Specify whether the code generator is to declare a signal data object as global data in the generated code.

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.

Command-Line Information: Real-Time Workshop Pane: Memory Sections Tab (Continued)

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

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.

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ERTFirstTimeCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to control inclusion of the `firstTime` argument in the *`model`*`_initialize` function generated for a Simulink model. Default is `off` for custom and non-ERT targets and `on` for ERT targets.
`IncludeERTFirstTime` (ERT) string - **`off`**, `on`	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.

Index

A

B

C

D

E

F

G

I

M

P

R

S

T