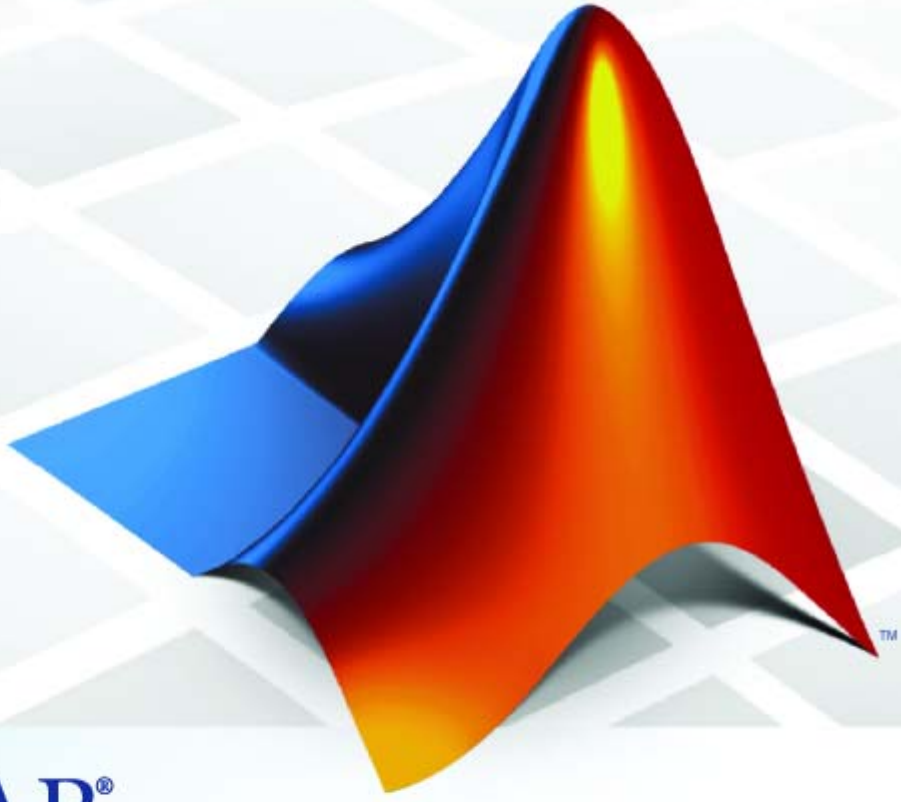
Real-Time Workshop® Embedded Coder™ 5
Reference



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)

Contents

1 Function Reference

AUTOSAR — Methods of arxml.importer 1-2

AUTOSAR — Methods of RTW.AutosarInterface 1-2

Function Prototype Control 1-4

Model Entry Points 1-6

System Target File Callback Interface 1-7

Target Function Library Table Creation 1-8

2 Functions — Alphabetical List

3 Block Reference

Configuration Wizards 3-2

Module Packaging 3-3

4 Blocks — Alphabetical List

5 Configuration Parameters

Real-Time Workshop Pane: Code Style **5-2**
Code Style Tab Overview **5-4**
Parentheses level **5-5**
Preserve operand order in expression **5-7**
Preserve condition expression in if statement **5-8**

Real-Time Workshop Pane: Templates **5-10**
Templates Tab Overview **5-12**
Code templates: Source file (*.c) template **5-13**
Code templates: Header file (*.h) template **5-14**
Data templates: Source file (*.c) template **5-15**
Data templates: Header file (*.h) template **5-16**
File customization template **5-17**
Generate an example main program **5-18**
Target operating system **5-20**

Real-Time Workshop Pane: Data Placement **5-22**
Data Placement Tab Overview **5-24**
Data definition **5-25**
Data definition filename **5-27**
Data declaration **5-29**
Data declaration filename **5-31**
#include file delimiter **5-32**
Module naming **5-33**
Module name **5-35**
Signal display level **5-37**
Parameter tune level **5-39**

Real-Time Workshop Pane: Data Type Replacement . . **5-41**
Data Type Replacement Tab Overview **5-43**
Replace data type names in the generated code **5-44**
Replacement Name: double **5-46**
Replacement Name: single **5-48**

Replacement Name: int32 5-50
Replacement Name: int16 5-52
Replacement Name: int8 5-54
Replacement Name: uint32 5-56
Replacement Name: uint16 5-58
Replacement Name: uint8 5-60
Replacement Name: boolean 5-62
Replacement Name: int 5-64
Replacement Name: uint 5-66
Replacement Name: char 5-68

Real-Time Workshop Pane: Memory Sections **5-70**
Memory Sections Tab Overview 5-72
Package 5-73
Refresh package list 5-75
Initialize/Terminate 5-76
Execution 5-77
Constants 5-78
Inputs/Outputs 5-80
Internal data 5-82
Parameters 5-84
Validation results 5-86

Real-Time Workshop Pane: AUTOSAR Code Generation Options **5-87**
AUTOSAR Code Generation Options Tab Overview 5-89
Generate XML file from schema version 5-90
Configure AUTOSAR Interface 5-91

Parameter Reference **5-92**
Recommended Settings Summary 5-92
Parameter Command-Line Information Summary 5-101

Index

Function Reference

AUTOSAR — Methods of arxml.importer (p. 1-2)	Control AUTOSAR component configuration for import into Simulink® models
AUTOSAR — Methods of RTW.AutosarInterface (p. 1-2)	Control AUTOSAR component configuration for code generation and XML file export from Simulink models
Function Prototype Control (p. 1-4)	Control step function prototypes in generated code for ERT-based Simulink models
Model Entry Points (p. 1-6)	Access entry points in generated code for ERT-based Simulink models
System Target File Callback Interface (p. 1-7)	Control Real-Time Workshop® configuration options in callbacks for ERT-based custom targets
Target Function Library Table Creation (p. 1-8)	Create function replacement tables that make up Real-Time Workshop target function libraries (TFLs)

AUTOSAR — Methods of arxml.importer

`createComponentAsModel`	Create AUTOSAR atomic software component as Simulink® model
`createComponentAsSubsystem`	Create AUTOSAR atomic software component as Simulink atomic subsystem
`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 — Methods of RTW.AutosarInterface

`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
`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
`getIOInterfaceName`	Get I/O interface name
`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

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`	Get argument category for Simulink model port from model-specific C function prototype
`getArgName`	Get argument name for Simulink model port from model-specific C function prototype
`getArgPosition`	Get argument position for Simulink model port from model-specific C function prototype
`getArgQualifier`	Get argument type qualifier for Simulink model port from model-specific C function prototype
`getDefaultConf (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`	Get number of function arguments from model-specific C function prototype
`getPreview`	Get model-specific C function prototype code preview
`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`	Set argument category for Simulink model port in model-specific C function prototype
`setArgName`	Set argument name for Simulink model port in model-specific C function prototype
`setArgPosition`	Set argument position for Simulink model port in model-specific C function prototype
`setArgQualifier`	Set argument type qualifier for Simulink model port in model-specific C function prototype
`setFunctionName`	Set function name in model-specific C function prototype

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
`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

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 Step Function** button on the **Interface** pane of the Configuration Parameters dialog box to bring up the Model Step Function 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_step 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 (Function Prototype Control)

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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

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

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

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

createAndSetCImplementationReturn

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

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`

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`

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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

getArgName

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_step Function Prototypes" 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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

getArgQualifier

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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

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`

getComponentNames

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`

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`.

getDefaultConf (AUTOSAR)

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.

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_step 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

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`

getFunctionName

Purpose

Get function name from model-specific C function prototype

Syntax

```
fcnName = getFunctionName(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

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 function described by the specified model-specific C function prototype.

See Also

"Controlling model_step Function Prototypes" 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

obj

Handle to the function control object associated with the input model.

modelName

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

Returns

String specifying the handle to the model. If the model does not have any function control object, then just returns an empty string.

Description

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

See Also

"Controlling model_step 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

`getIOCategory` 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`

getIOInterfaceName

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`

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_step 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`.

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`.

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)
```

Arguments

obj

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

Returns

String specifying the function prototype for the *model*`_step` function.

Description

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

See Also

"Controlling model_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

getTflArgFromString

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

importer

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`

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

Purpose

Set event flags for multirate, multitasking operation before calling *model*_step for ERT-based Simulink® model

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

The Real-Time Workshop Embedded Coder software generates the *model*`_SetEventsForThisBaseStep` utility function only for multirate, multitasking models.

The *model*`_SetEventsForThisBaseStep` 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. See "Multirate Multitasking Operation" in the Real-Time Workshop Embedded Coder documentation for further information.

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.

See Also

`model_initialize`, `model_step`, `model_terminate`

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

Purpose

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

Syntax

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

Arguments

`tid`

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

Calling Interfaces

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

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

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

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

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

For greater control over the *model*_step function prototype, you can use the **Configure Step Function** button on the **Interface** pane to launch a Model Interface dialog box (see "Model Interface Dialog Box" 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-step function` and `Model specific C prototype`), 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_step 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

registerCFunctionEntry

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:

abs	ceil	floor	sinh
acos	cos	log	sqrt
asin	cosh	log10	tan
atan	exp	sin	tanh

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

- 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

runValidation (AUTOSAR)

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 following checks:

1. `step`, `init runnable` names and `event` names are valid AUTOSAR short name identifiers (see definition 1 following).
2. Storage class of root I/O ports is `auto`.
3. AUTOSAR port, interface, and data element names are valid AUTOSAR short name identifiers (see definition 1 following).
4. AUTOSAR XML options for the component name, internal behavior name, and implementation name are valid AUTOSAR path and shortname identifiers (see definition 2 following).
5. AUTOSAR XML options for the interface package name and data type package name are valid AUTOSAR path identifiers (see definition 3 following).
6. I/O is 1D or scalar.

7 Simulink® ports may have duplicated AUTOSAR port names, however the AUTOSAR Interface name must also be the same.

8 A Simulink inport and an outport cannot have the same AUTOSAR port name.

9 For any duplicated AUTOSAR port name and AUTOSAR Interface name, the data element names must be unique.

10 Model contains no custom code blocks.

11 No absolute time.

12 No continuous time.

13 No non-inlined S-functions.

14 No non-finite numbers.

15 No complex numbers.

16 No multitasking.

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.

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, `message` contains 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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

setArgCategory

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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

Purpose

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

Syntax

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

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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

setArgPosition

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_step Function Prototypes" in the Real-Time Workshop® Embedded Coder™ documentation

Purpose

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

Syntax

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

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_step Function Prototypes" 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)
```

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.

Description

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

See Also

"Controlling model_step 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 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:

`abs`	`ceil`	`floor`	`sinh`
`acos`	`cos`	`log`	`sqrt`
`asin`	`cosh`	`log10`	`tan`
`atan`	`exp`	`sin`	`tanh`

`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 `''`.

Description

The `setTflCFunctionEntryParameters` function sets specified parameters for a function entry in a TFL table.

Example

In the following example, the `setTflCFunctionEntryParameters` function is used to set specified parameters for a TFL function entry for `sqrt`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                  'Key',                      'sqrt', ...
                                  'Priority',                 100, ...
                                  'ImplementationName',       'sqrt', ...
                                  'ImplementationHeaderFile', '<math.h>' );
```

See Also

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

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

"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`.

`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`.

`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 `''`.

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

slConfigUIGetVal

Purpose

Return current value for custom target configuration option

Syntax

```
value = slConfigUIGetVal(hDlg, hSrc, 'OptionName')
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

Returns

Current value of the specified option. The data type of the return value depends on the data type of the option.

Description

The `slConfigUIGetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUIGetVal` to read the current value of a specified target option.

Example

In the following example, the `slConfigUIGetVal` function returns the value of the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
```

```
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

    slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
    slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUISetEnabled`, `slConfigUISetVal`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop® Embedded Coder™ documentation

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

slConfigUISetEnabled

Purpose

Enable or disable custom target configuration option

Syntax

```
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', true)
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', false)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`true`
: Specifies that the option should be enabled.

`false`
: Specifies that the option should be disabled.

Description

The `slConfigUISetEnabled` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetEnabled` to enable or disable a specified target option.

Example

In the following example, the `slConfigUISetEnabled` function disables the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
```

```
        '  Uncheck and disable "Terminate function required".']);

disp(['Value of IncludeMdlTerminateFcn was ', ...
      slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUIGetVal`, `slConfigUISetVal`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop® Embedded Coder™ documentation

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

slConfigUISetVal

Purpose

Set value for custom target configuration option

Syntax

```
slConfigUISetVal(hDlg, hSrc, 'OptionName', OptionValue)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`OptionValue`
: Value to be set for the specified option.

Description

The `slConfigUISetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetVal` to set the value of a specified target option.

Example

In the following example, the `slConfigUISetVal` function sets the value `'off'` for the **Terminate function required** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);
```

```
slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUIGetVal`, `slConfigUISetEnabled`

"Defining and Displaying Custom Target Options" in the Real-Time Workshop® Embedded Coder™ documentation

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

syncWithModel

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 (p. 3-2)	Automatically update configuration of parent Simulink® model
Module Packaging (p. 3-3)	Create potential Simulink data objects

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

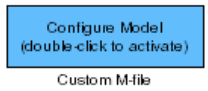
Purpose

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

Library

Configuration Wizards

Description



Custom M-file

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

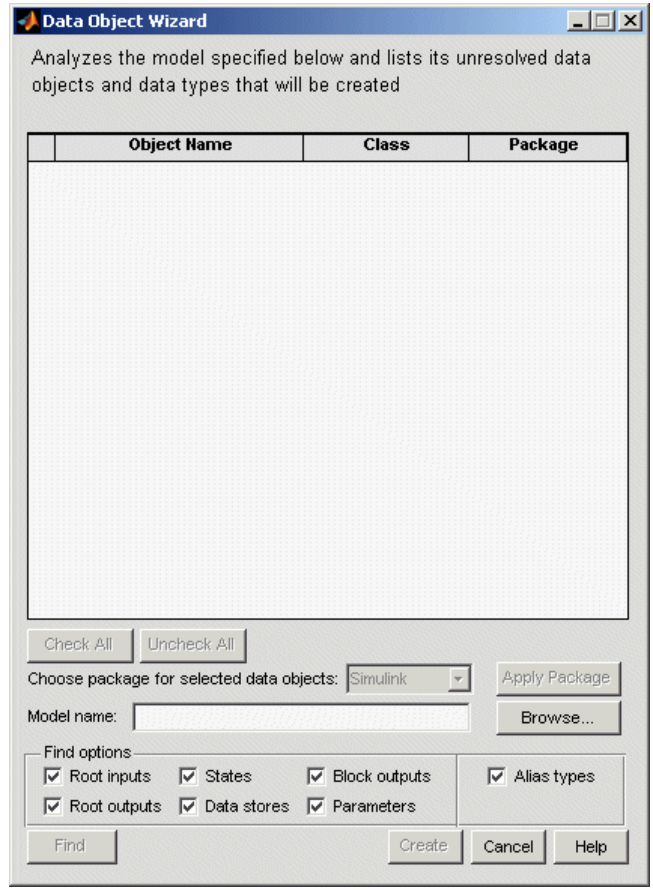
Simulink® data object wizard for creating potential Simulink data objects

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)

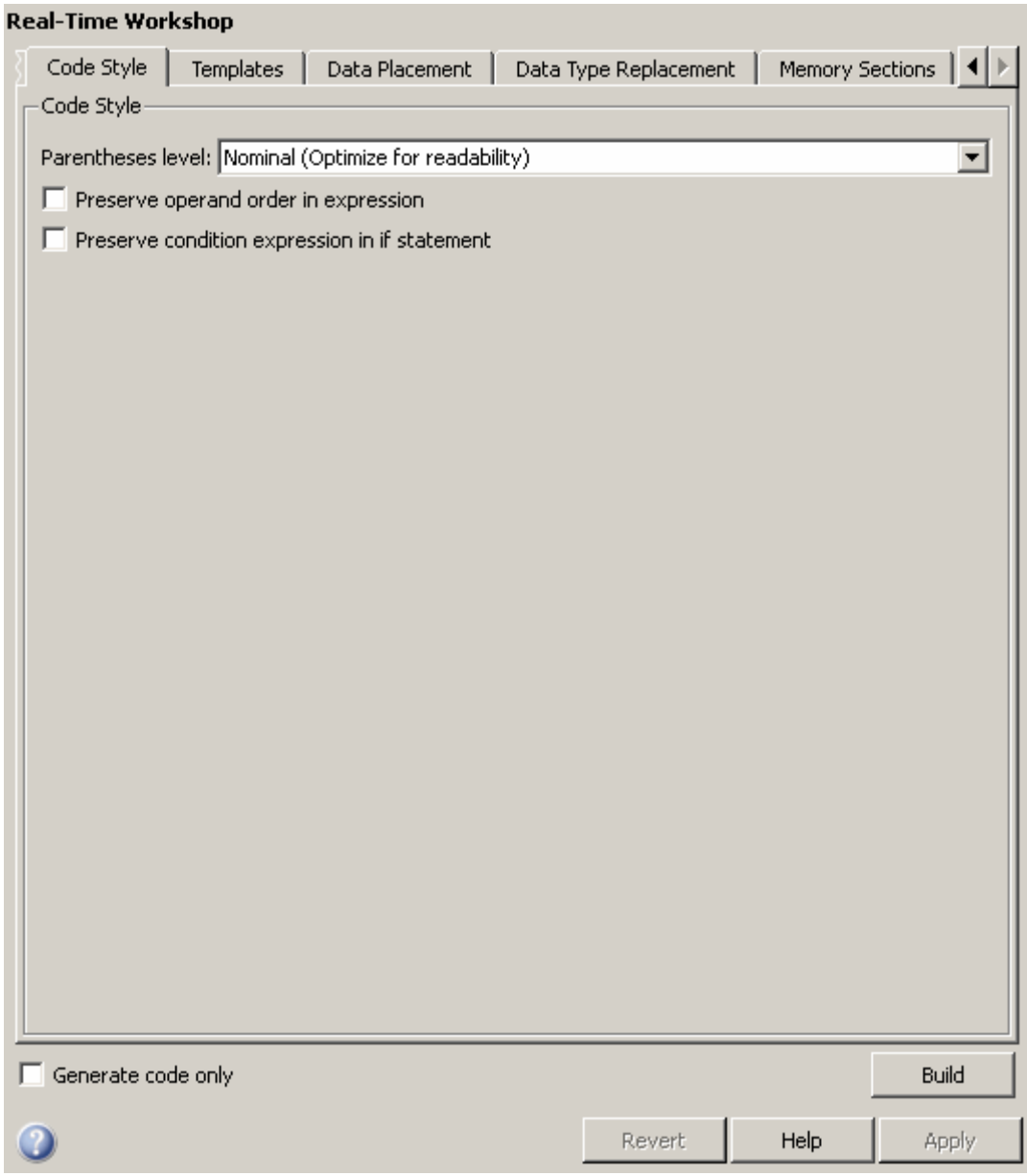
"Optimizing Your Model with Configuration Wizard Blocks and Scripts" in the Real-Time Workshop® Embedded Coder™ documentation

GRT (optimized for fixed/floating-point)

Configuration Parameters

Real-Time Workshop Pane: Code Style (p. 5-2)	Parameters for controlling the automatically generated code style for readability
Real-Time Workshop Pane: Templates (p. 5-10)	Parameters for customizing generated code organization
Real-Time Workshop Pane: Data Placement (p. 5-22)	Parameters for specifying data placement in the generated code
Real-Time Workshop Pane: Data Type Replacement (p. 5-41)	Parameters for replacing built-in data type names with user-defined replacement data type names in the generated code
Real-Time Workshop Pane: Memory Sections (p. 5-70)	Parameters for inserting comments and pragmas into the generated code
Real-Time Workshop Pane: AUTOSAR Code Generation Options (p. 5-87)	Parameters for controlling AUTOSAR code generation options.
Parameter Reference (p. 5-92)	Summary of code generation parameters for tuning model and target configurations

Real-Time Workshop Pane: Code Style



In this section...
“Code Style Tab Overview” on page 5-4
“Parentheses level” on page 5-5
“Preserve operand order in expression” on page 5-7
“Preserve condition expression in if statement” on page 5-8

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
```

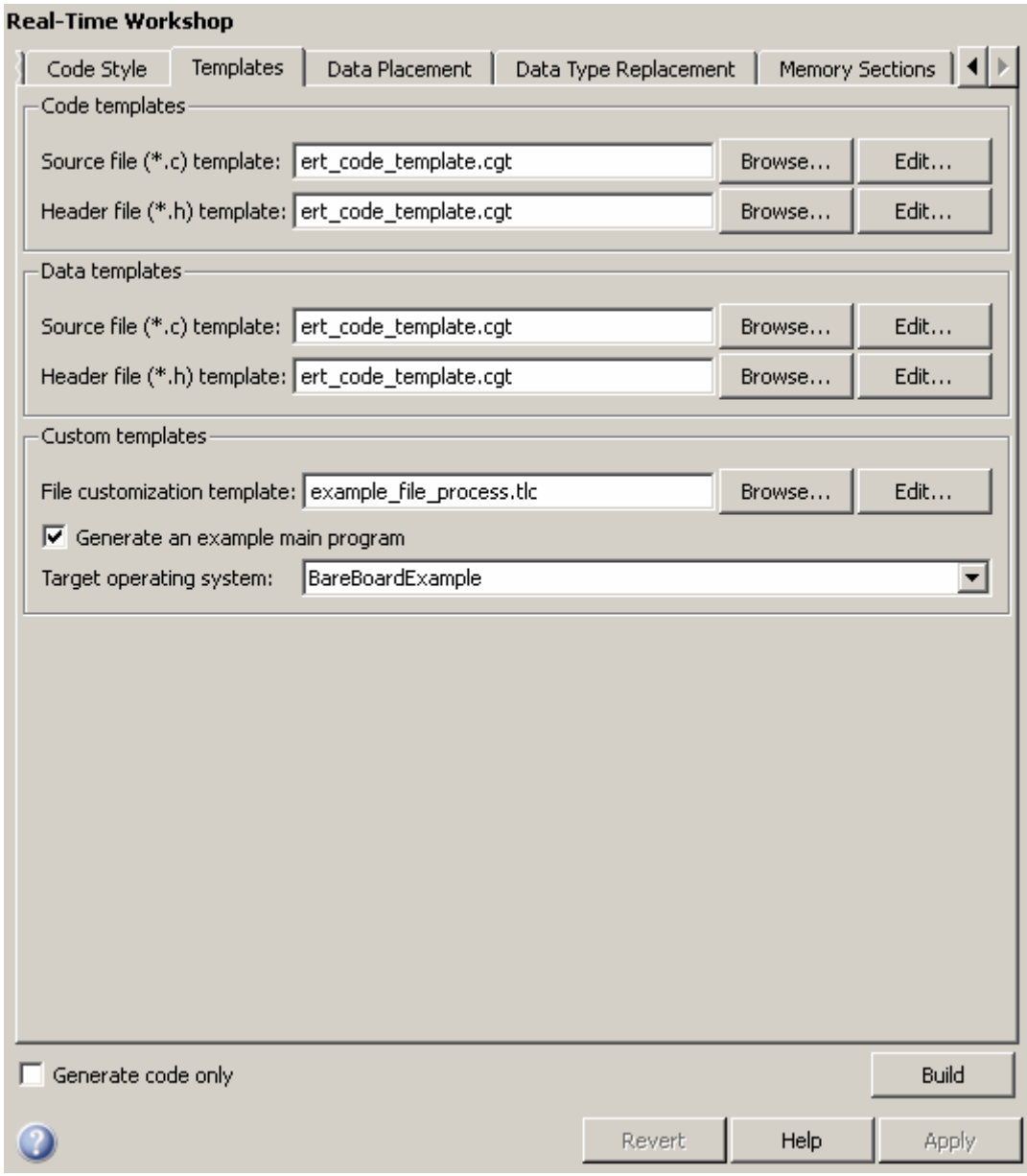
Command-Line Information

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" on page 5-12
"Code templates: Source file (*.c) template" on page 5-13
"Code templates: Header file (*.h) template" on page 5-14
"Data templates: Source file (*.c) template" on page 5-15
"Data templates: Header file (*.h) template" on page 5-16
"File customization template" on page 5-17
"Generate an example main program" on page 5-18
"Target operating system" on page 5-20

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'`

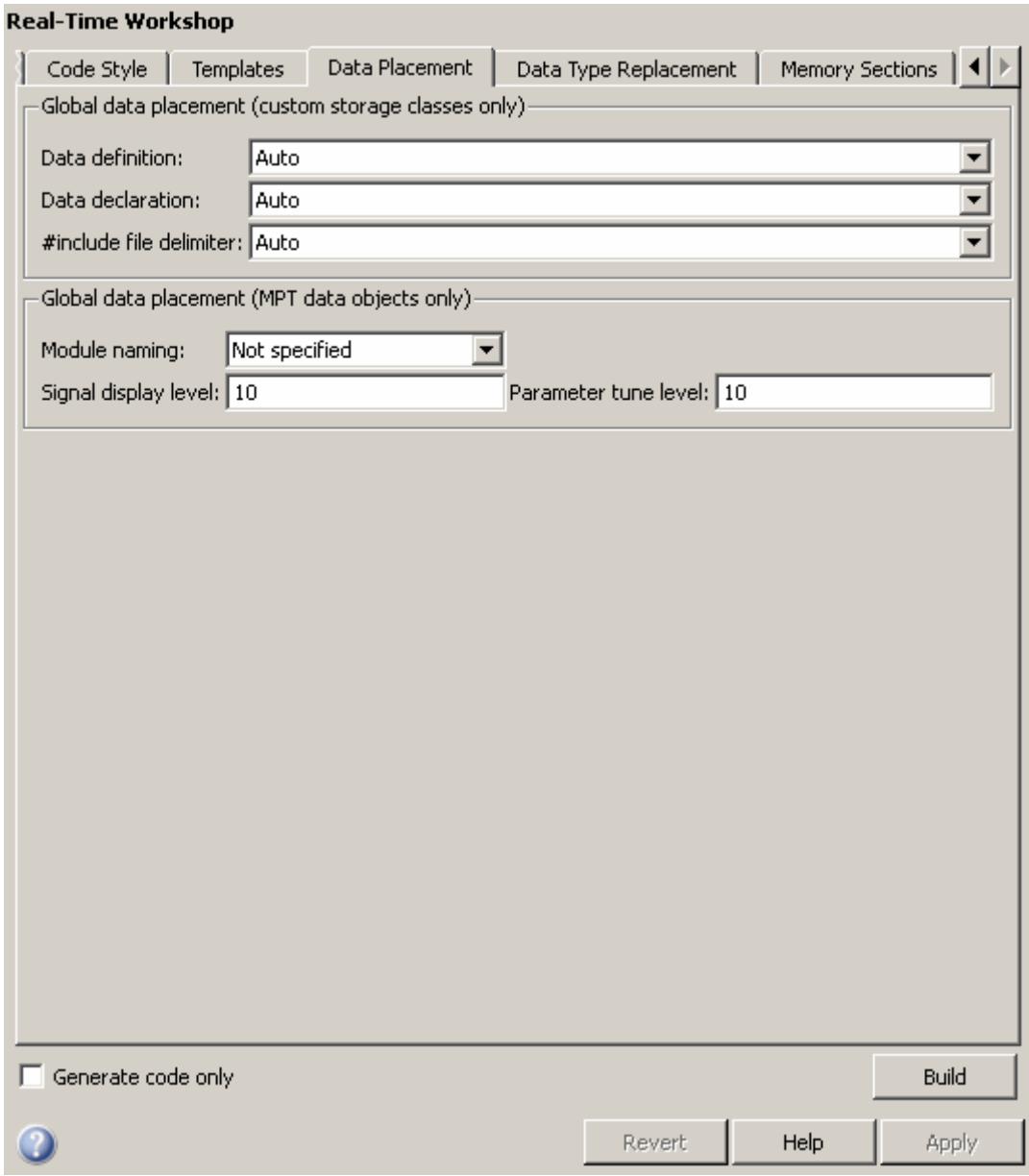
Recommended Settings

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" on page 5-24
"Data definition" on page 5-25
"Data definition filename" on page 5-27
"Data declaration" on page 5-29
"Data declaration filename" on page 5-31
"#include file delimiter" on page 5-32
"Module naming" on page 5-33
"Module name" on page 5-35
"Signal display level" on page 5-37
"Parameter tune level" on page 5-39

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`

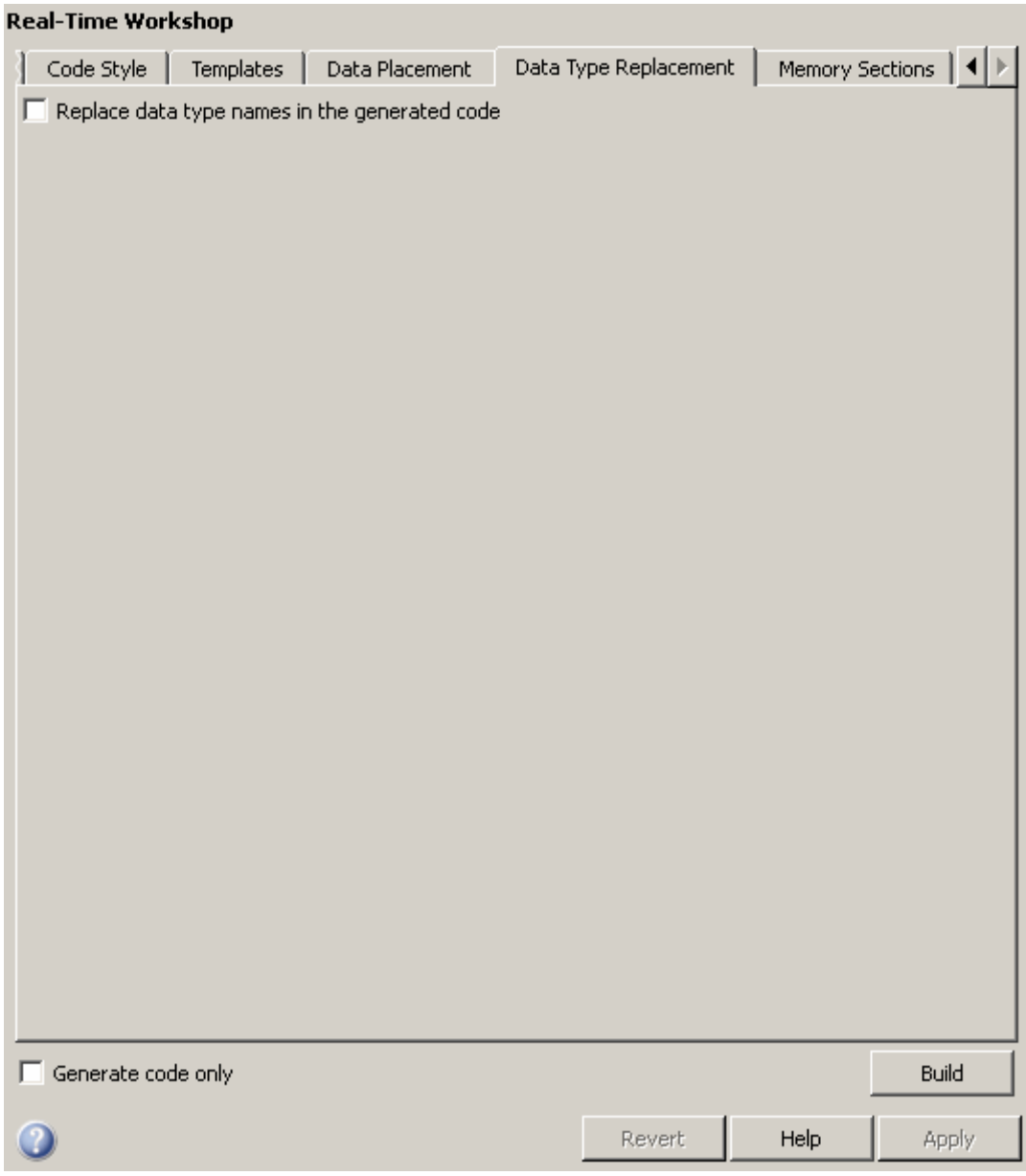
Recommended Settings

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” on page 5-43
“Replace data type names in the generated code” on page 5-44
“Replacement Name: double” on page 5-46
“Replacement Name: single” on page 5-48
“Replacement Name: int32” on page 5-50
“Replacement Name: int16” on page 5-52
“Replacement Name: int8” on page 5-54
“Replacement Name: uint32” on page 5-56
“Replacement Name: uint16” on page 5-58
“Replacement Name: uint8” on page 5-60
“Replacement Name: boolean” on page 5-62
“Replacement Name: int” on page 5-64
“Replacement Name: uint” on page 5-66
“Replacement Name: char” on page 5-68

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

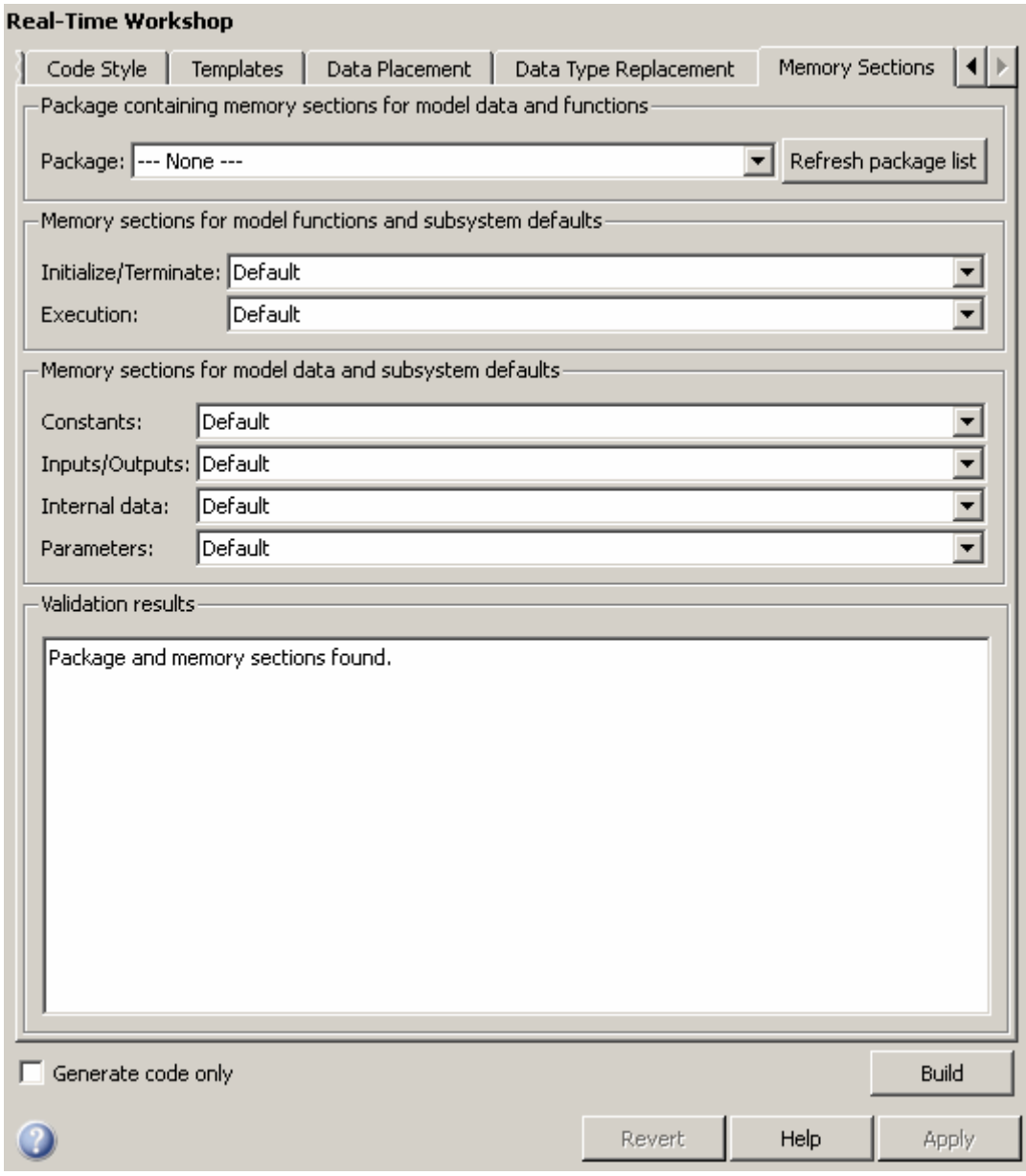
Application	Setting
Debugging	No impact
Traceability	Any valid string

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" on page 5-72
"Package" on page 5-73
"Refresh package list" on page 5-75
"Initialize/Terminate" on page 5-76
"Execution" on page 5-77
"Constants" on page 5-78
"Inputs/Outputs" on page 5-80
"Internal data" on page 5-82
"Parameters" on page 5-84
"Validation results" on page 5-86

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.

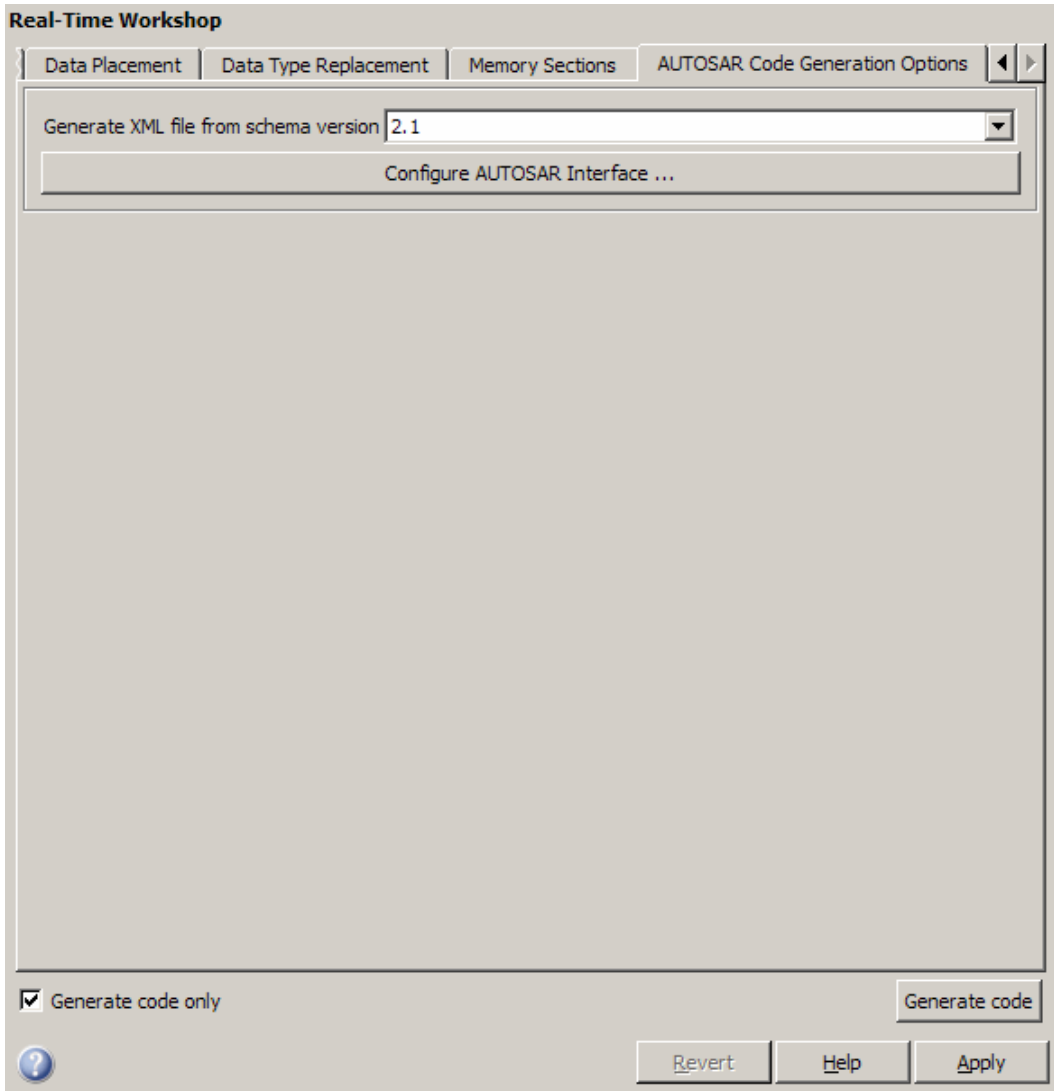
Settings

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” on page 5-89
“Generate XML file from schema version” on page 5-90
“Configure AUTOSAR Interface” on page 5-91

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 — Methods of RTW.AutosarInterface" on page 1-2
- "AUTOSAR — Methods of arxml.importer" 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

- "AUTOSAR Model Interface Dialog Box"
- "Generating Code That Complies with AUTOSAR Standards"

Parameter Reference

In this section...

"Recommended Settings Summary" on page 5-92

"Parameter Command-Line Information Summary" on page 5-101

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 state zero initialization	No impact	No impact	On	Off	Off
Use memset to initialize floats and doubles to 0.0	No impact	No impact	On	No impact	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
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`
Suppress error status in real-time model data structure	Off	No impact	On	On	Off
Single output/update function	On	On	On	On	On
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
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)`
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`
File customization template	No impact	No impact	No impact	No impact	`example_file_-process.tlc`
Generate an example main program	No impact	No impact	No impact	No impact	On
Target operating system	No impact	No impact	No impact	No impact	`BareBoard-Example`

Mapping of Application Requirements to the Real-Time Workshop Pane: 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
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`
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 bit sets for storing boolean data**	Use bit sets for storing Boolean data.
`InitFltsAndDblsToZero` (ERT) `off`, **`on`**	**Optimization > Use memset to initialize floats and doubles to 0.0**	Optimize initialization of storage for `float` and `double` values. Set this option if the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern 0.
`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.

Command-Line Information: Optimization Pane (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`OptimizeModelRefInitCode` (ERT) **`off`**, `on`	**Optimization > Optimize initialization code for model reference**	Suppress generation of initialization code to accommodate the case where this model is referred to by a subsystem that resets its states when enabled. Select this option if the model will never be referred to by such a subsystem. The Simulink engine reports an error if this constraint is violated, in which case you can disable this optimization.
`StateBitsets` (Stateflow) **`off`**, `on`	**Optimization > Use bit sets for storing state configuration**	Use bit sets for storing state configuration.
`UseTempVars` (Stateflow) **`off`**, `on`	**Optimization > Minimize array reads using temporary variables**	Minimize array reads in global memory by using temporary variables.
`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 state zero initialization**	Suppress code that initializes global data structures (for example, block I/O data structures) to zero.

Command-Line Information: Real-Time Workshop Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IgnoreCustomStorageClasses` (ERT) *string* - `off`, **`on`**	**Real-Time Workshop > General > Ignore custom storage classes**	Treat custom storage classes as `'Auto'`.

Command-Line Information: Real-Time Workshop Pane: Report Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateTraceInfo` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Report > Model-to-code**	Includes model-to-code traceability support in the generated HTML report.
`IncludeHyperlinkInReport` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Report > Code-to-model**	Link code segments to the corresponding object in the model. This option increases code generation time for large models.
`GenerateTraceReport` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Report > Eliminated / virtual blocks**	Include summary of eliminated and virtual blocks in Code Generation report.
`GenerateTraceReportSl` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Report > Traceable Simulink blocks**	Include summary of Simulink blocks in Code Generation report.
`GenerateTraceReportSf` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Report > Traceable Stateflow objects**	Include summary of Stateflow objects in Code Generation report.
`GenerateTraceReportEml` (ERT) *string* - **`off`**, `on`	**Real-Time Workshop > Report > Traceable Embedded MATLAB functions**	Include summary of Embedded MATLAB™ functions in Code Generation report.

Command-Line Information: Real-Time Workshop Pane: Comments Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomCommentsFcn` (ERT) *`string`* -	**Real-Time Workshop > Comments > Custom comments function**	Specify the filename of the 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.
`InsertBlockDesc` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Simulink block descriptions**	Insert the contents of the **Description** field from the Block Parameters dialog box into the generated code as a comment.
`ReqsInCode` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Requirements in block comments**	Include specified requirements in the generated code as a comment.
`SFDataObjDesc` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Stateflow object descriptions**	Insert Stateflow object descriptions into the generated code as a comment.
`SimulinkDataObjDesc` (ERT) *`string`* - **`off`**, `on`	**Real-Time Workshop > Comments > Simulink data object descriptions**	Insert Simulink data object descriptions into the generated code as comments.

Command-Line Information: Real-Time Workshop Pane: Symbols Tab

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

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrGlobalVar` (ERT) *`string`* - **`$R$N$M`**	**Real-Time Workshop > Symbols > Global variables**	Specify a symbol format rule for global variables. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object
`CustomSymbolStrMacro` (ERT) *`string`* - **`$R$N$M`**	**Real-Time Workshop > Symbols > Constant macros**	Specify a symbol format rule for constant macros. The rule can contain valid C identifier characters and the following macros: $M - Mangle $R - Root model name $N - Name of object
`CustomSymbolStrTmpVar` (ERT) *`string`* - **`$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

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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`.
`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'`.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncHierarchyInIds` (ERT) **`off`**, `on`	**Real-Time Workshop > Symbols > Include system hierarchy number in identifiers**	Include the system hierarchy number in variable identifiers. For example, `'s3_'` is the system hierarchy number in `rtB.s3_signame` for a block output signal named `'signame'`. Including the system hierarchy number in identifiers improves the traceability of generated code. To locate the subsystem in which the identifier resides, type `hilite_system('<S3>')` at the MATLAB prompt. The argument specified with `hilite_system` requires an uppercase `S`.
`InlinedPrmAccess` (ERT) string - **`Literals`**, `Macros`	**Real-Time Workshop > Symbols > Generate scalar inlined parameters as**	Specify whether inlined parameters are coded as numeric constants or macros. Specify `Macros` for more efficient code.
`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.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`ParamNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > Parameter naming**	Select a rule that changes spelling of all parameter names.
`PrefixModelToSubsysFcnNames` (ERT) `off`, **`on`**	**Real-Time Workshop > Symbols > Prefix model name to global identifiers**	Add the model name as a prefix to subsystem function names for all code formats. When appropriate for the code format, also add the model name as a prefix to top-level functions and data structures. This prevents compiler errors due to name clashes when combining multiple models.
`SignalNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Real-Time Workshop > Symbols > Signal naming**	Specify a rule the code generator is to use that changes spelling of all signal names.

Command-Line Information: Real-Time Workshop Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CombineOutputUpdateFcns` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > Single output/update function**	Generate a model's output and update routines into a single-step function.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`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.
`MatFileLogging` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > MAT-file logging**	Generate code that logs data to a MATLAB `.mat` file.
`MultiInstanceErrorCode` (ERT) string - `None`, `Warning`, **`Error`**	**Real-Time Workshop > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.
`MultiInstanceERTCode` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`RootIOFormat` (ERT) string - **`Individual arguments`**, `Structure reference`	**Real-Time Workshop > Interface > Pass root-level I/O as**	Specify how the code generator is to pass root-level I/O data into a reusable function.
`SupportAbsoluteTime` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > absolute time**	Support absolute time in the generated code. Blocks such as the Discrete Integrator might require absolute time.
`SupportComplex` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > complex numbers**	Support complex data types in the generated code.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SupportContinuousTime` (ERT) string - `off`, **`on`**	**Real-Time Workshop > Interface > continuous time**	Support continuous time in the generated code. This allows blocks to be configured with a continuous sample time. Not available if `SuppressErrorStatus` is `on`.
`SupportNonFinite` (ERT) string - **`off`**, `on`	**Real-Time Workshop > Interface > nonfinite numbers**	Support nonfinite values (`inf`, `nan`, `-inf`) in the generated code. This option is forced on when `SupportNonInlinedSFcns` is on.
`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.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`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.

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`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.

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
`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.
`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.
`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

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

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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
`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
`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 step functions that are generated for a Simulink model. Default is `off` for non-ERT targets and `on` for ERT targets.

Index

A

addAdditionalHeaderFile function 2-2
addAdditionalIncludePath function 2-4
addAdditionalLinkObj function 2-6
addAdditionalLinkObjPath function 2-7
addAdditionalSourceFile function 2-8
addAdditionalSourcePath function 2-10
addArgConf function 2-12
addConceptualArg function 2-14
addEntry function 2-16
addIOConf AutosarInterface method 1-2 2-18
arxml.importer method 1-2 2-39
attachToModel AutosarInterface method 1-2 2-19
attachToModel function 2-20
AUTOSAR 1-2 to 1-3 2-19 2-39 2-50 2-56 to 2-57 2-61 to 2-64 2-66 to 2-67
  addIOConf 1-2 2-18
  createComponentAsSubsystem 1-2 2-41
  getComponentName 1-2 2-47
  getComponentNames 1-2 2-48
  getDataTypePackageName 1-2 2-49
  getDependencies 1-2 2-52
  getFile 1-2 2-53
  getImplementationName 1-2 2-60
  getInterfacePackageName 1-2 2-58
  getInternalBehaviorName 1-3 2-59
  getPortDefaultConf 1-3 2-68
  importer 1-2 2-72
  runValidation 1-3 2-86
  setComponentName 1-3 2-94
  setDependencies 1-2 2-95
  setFile 1-2 2-96
  setInitEventName 1-3 2-98
  setInitRunnableName 1-3 2-99
  setIOAutosarPortName 1-3 2-100
  setIODataAccessMode 1-3 2-101
  setIODataElement 1-3 2-102
  setIOInterfaceName 1-3 2-103
  setPeriodicEventName 1-3 2-104
  setPeriodicRunnableName 1-3 2-105
  syncWithModel 1-3 2-124
AUTOSAR Code Generation Options pane 5-87

B

blocks
  Custom M-file 4-2
  Data Object Wizard 4-4
  ERT (optimized for fixed-point) 4-6
  ERT (optimized for floating-point) 4-8
  GRT (debug for fixed/floating-point) 4-10
  GRT (optimized for fixed/floating-point) 4-12

C

Code Style pane 5-2
configuration parameters
  code generation 5-101
  impacts of settings 5-92
  pane
    Autosar Schema Version 5-90
    gui name 5-91
  Real-Time Workshop pane: AUTOSAR Code Generation Options 5-89
  Real-Time Workshop pane: Code Style 5-4
  Real-Time Workshop pane: Data Placement 5-24
  Real-Time Workshop pane: Data Type Replacement 5-43
  Real-Time Workshop pane: Memory Sections 5-72
  Real-Time Workshop pane: Templates 5-12
Configuration Parameters dialog box
  Code Style pane
    Parentheses level 5-5
    Preserve condition expression in if statement 5-8
    Preserve operand order in expression 5-7

Data Placement pane
Data declaration 5-29
Data declaration filename 5-31
Data definition 5-25
Data definition filename 5-27
#include file identifier 5-32
Module name 5-35
Module naming 5-33
Parameter tune level 5-39
Signal display level 5-37
Data Type Replacement pane
boolean Replacement Name 5-62
char Replacement Name 5-68
double Replacement Name 5-46
int Replacement Name 5-64
int16 Replacement Name 5-52
int32 Replacement Name 5-50
int8 replacement name 5-54
Replace data type names in the generated code 5-44
single Replacement Name 5-48
uint Replacement Name 5-66
uint16 Replacement Name 5-58
uint32 Replacement Name 5-56
uint8 Replacement Name 5-60
Memory Sections pane
Constants 5-78
Execution 5-77
Initialize/Terminate 5-76
Inputs/Outputs 5-80
Internal data 5-82
Package 5-73
Parameters 5-84
Refresh package list 5-75
Validation results 5-86
Templates pane
code templates: Header file (*.h) template 5-14
code templates: Source file (*.c) template 5-13
data templates: Header file (*.h) template 5-16
data templates: Source file (*.c) template 5-15
File customization template 5-17
Generate an example main program 5-18
Target operating system 5-20
`copyConceptualArgsToImplementation` function 2-21
`createAndAddConceptualArg` function 2-23
`createAndAddImplementationArg` function 2-29
`createAndSetCImplementationReturn` function 2-34
`createComponentAsSubsystem` arxml.importer method 1-2 2-41
Custom M-file block 4-2

D

Data Object Wizard block 4-4
Data Placement pane 5-22
Data Type Replacement pane 5-41

E

ERT (optimized for fixed-point) block 4-6
ERT (optimized for floating-point) block 4-8

F

function prototype control
`addArgConf` 2-12
`attachToModel` 2-20
`getArgCategory` 2-43
`getArgName` 2-44
`getArgPosition` 2-45

getArgQualifier 2-46
getDefaultConf 2-51
getFunctionName 2-54
getNumArgs 2-65
getPreview 2-69
RTW.getFunctionSpecification 2-55
runValidation 2-89
setArgCategory 2-90
setArgName 2-91
setArgPosition 2-92
setArgQualifier 2-93
setFunctionName 2-97

G

getArgCategory function 2-43
getArgName function 2-44
getArgPosition function 2-45
getArgQualifier function 2-46
getComponentName AutosarInterface method 1-2 2-47
getComponentNames arxml.importer method 1-2 2-48
getDataTypePackageName AutosarInterface method 1-2 2-49
getDefaultConf AutosarInterface method 1-2 2-50
getDefaultConf function 2-51
getDependencies arxml.importer method 1-2 2-52
getFile arxml.importer method 1-2 2-53
getFunctionName function 2-54
getImplementationName AutosarInterface method 1-2 2-60
getInitEventName AutosarInterface method 1-2 2-56
getInitRunnableName AutosarInterface method 1-2 2-57
getInterfacePackageName AutosarInterface method 1-2 2-58
getInternalBehaviorName AutosarInterface method 1-3 2-59
getIOAutosarPortName AutosarInterface method 1-3 2-61
getIODataAccessMode AutosarInterface method 1-3 2-62
getIODataElement AutosarInterface method 1-3 2-63
getIOInterfaceName AutosarInterface method 1-3 2-64
getNumArgs function 2-65
getPeriodicEventName AutosarInterface method 1-3 2-66
getPeriodicRunnableName AutosarInterface method 1-3 2-67
getPortDefaultConf AutosarInterface method 1-3 2-68
getPreview function 2-69
getTflArgFromString function 2-70
GRT (debug for fixed/floating-point) block 4-10
GRT (optimized for fixed/floating-point) block 4-12

I

importer arxml.importer method 1-2 2-72

M

Memory Sections pane 5-70
model entry points
model_initialize 2-73
model_SetEventsForThisBaseStep 2-75
model_step 2-76
model_terminate 2-79
model_initialize function 2-73
model_output function 2-77
model_SetEventsForThisBaseStep function 2-75
model_step function 2-76

model_terminate function 2-79
model_update function 2-77
models
    parameters for configuring 5-101

P

parameters
    for configuring model code generation and targets 5-101

R

registerCFunctionEntry function 2-80
registerCPromotableMacroEntry function 2-83
RTW.getFunctionSpecification function 2-55
runValidation AutosarInterface method 1-3 2-86
runValidation function 2-89

S

setArgCategory function 2-90
setArgName function 2-91
setArgPosition function 2-92
setArgQualifier function 2-93
setComponentName AutosarInterface method 1-3 2-94
setDependencies arxml.importer method 1-2 2-95
setFile arxml.importer method 1-2 2-96
setFunctionName function 2-97
setInitEventName AutosarInterface method 1-3 2-98
setInitRunnableName AutosarInterface method 1-3 2-99
setIOAutosarPortName AutosarInterface method 1-3 2-100
setIODataAccessMode AutosarInterface method 1-3 2-101
setIODataElement AutosarInterface method 1-3 2-102
setIOInterfaceName AutosarInterface method 1-3 2-103
setPeriodicEventName AutosarInterface method 1-3 2-104
setPeriodicRunnableName AutosarInterface method 1-3 2-105
setReservedIdentifiers function 2-106
setTflCFunctionEntryParameters function 2-109
setTflCOperationEntryParameters function 2-112
slConfigUIGetVal function 2-118
slConfigUISetEnabled function 2-120
slConfigUISetVal function 2-122
syncWithModel AutosarInterface method 1-3 2-124

T

targets
    parameters for configuring 5-101
Templates pane 5-10
TFL table creation
    addAdditionalHeaderFile 2-2
    addAdditionalIncludePath 2-4
    addAdditionalLinkObj 2-6
    addAdditionalLinkObjPath 2-7
    addAdditionalSourceFile 2-8
    addAdditionalSourcePath 2-10
    addConceptualArg 2-14
    addEntry 2-16
    copyConceptualArgsToImplementation 2-21
    createAndAddConceptualArg 2-23
    createAndAddImplementationArg 2-29
    createAndSetCImplementationReturn 2-34
    getTflArgFromString 2-70
    registerCFunctionEntry 2-80
    registerCPromotableMacroEntry 2-83

setReservedIdentifiers 2-106
setTflCFunctionEntryParameters 2-109
setTflCOperationEntryParameters 2-112