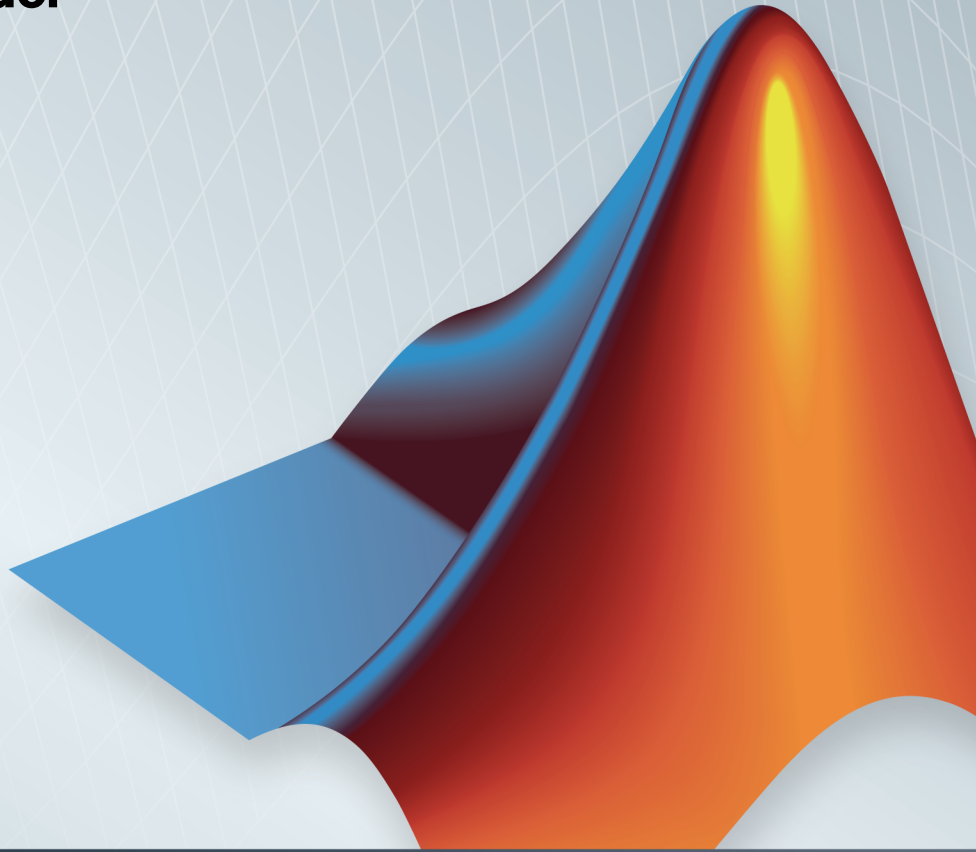


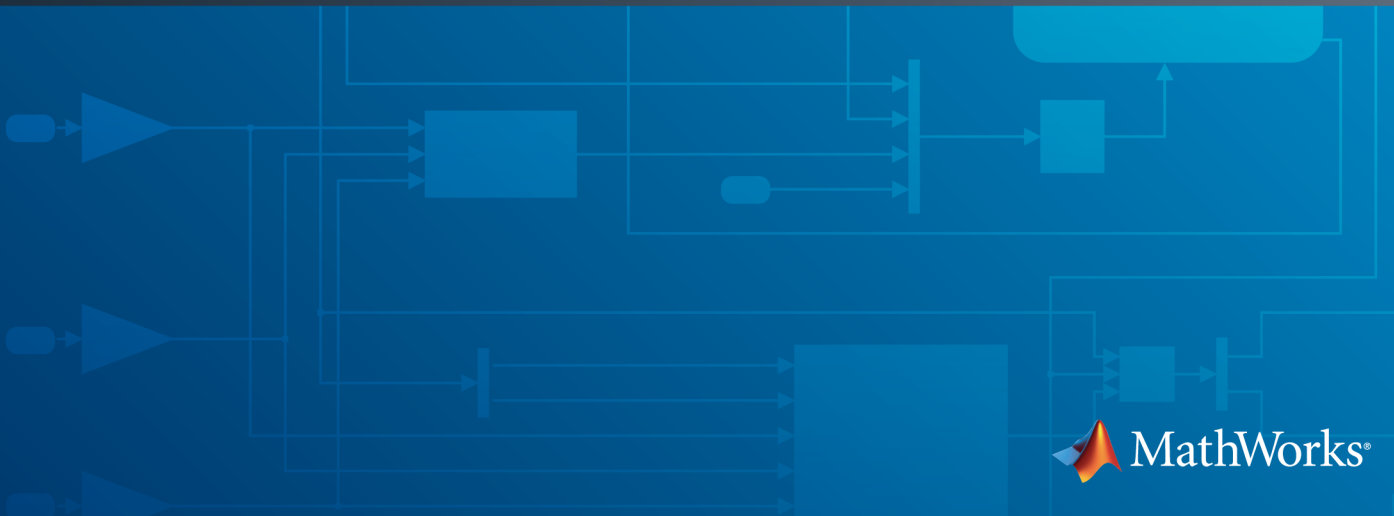
Embedded Coder[®]

Reference

R2014b



MATLAB[®] & SIMULINK[®]



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Embedded Coder[®] Reference

© COPYRIGHT 2011–2014 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	Revised for Version 6.1 (Release 2011b)
March 2012	Online only	Revised for Version 6.2 (Release 2012a)
September 2012	Online only	Revised for Version 6.3 (Release 2012b)
March 2013	Online only	Revised for Version 6.4 (Release 2013a)
September 2013	Online only	Revised for Version 6.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.6 (Release 2014a)
October 2014	Online only	Revised for Version 6.7 (Release 2014b)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Alphabetical List

1

Blocks — Alphabetical List

2

Configuration Parameters

3

Code Generation Pane: Verification	3-2
Code Generation: Verification Tab Overview	3-3
Measure task execution time	3-3
Measure function execution times	3-4
Workspace variable	3-5
Save options	3-6
Code coverage tool	3-7
Create block	3-9
Enable portable word sizes	3-10
Enable source-level debugging for SIL	3-11
Code Generation Pane: Code Style	3-13
Code Generation: Code Style Tab Overview	3-13
Parentheses level	3-14
Preserve operand order in expression	3-15
Preserve condition expression in if statement	3-16
Convert if-elseif-else patterns to switch-case statements ...	3-17
Preserve extern keyword in function declarations	3-19
Suppress generation of default cases for Stateflow switch statements if unreachable	3-20

Casting Modes	3-21
Indent style	3-23
Indent size	3-24
Code Generation Pane: Templates	3-26
Code Generation: Templates Tab Overview	3-26
Code templates: Source file (*.c) template	3-27
Code templates: Header file (*.h) template	3-27
Data templates: Source file (*.c) template	3-28
Data templates: Header file (*.h) template	3-29
File customization template	3-30
Generate an example main program	3-31
Target operating system	3-33
Code Generation Pane: Code Placement	3-35
Code Generation: Code Placement Tab Overview	3-35
Data definition	3-36
Data definition filename	3-37
Data declaration	3-38
Data declaration filename	3-39
Use owner from data object for data definition placement ..	3-40
#include file delimiter	3-41
Signal display level	3-42
Parameter tune level	3-43
File packaging format	3-44
Code Generation Pane: Data Type Replacement	3-47
Code Generation: Data Type Replacement Tab	3-48
Replace data type names in the generated code	3-48
Replacement Name: double	3-50
Replacement Name: single	3-51
Replacement Name: int32	3-53
Replacement Name: int16	3-54
Replacement Name: int8	3-55
Replacement Name: uint32	3-57
Replacement Name: uint16	3-58
Replacement Name: uint8	3-59
Replacement Name: boolean	3-61
Replacement Name: int	3-63
Replacement Name: uint	3-64
Replacement Name: char	3-66
Code Generation Pane: Memory Sections	3-68
Code Generation: Memory Sections Tab Overview	3-68

Package	3-69
Refresh package list	3-70
Initialize/Terminate	3-70
Execution	3-71
Shared utility	3-72
Constants	3-73
Inputs/Outputs	3-74
Internal data	3-75
Parameters	3-76
Validation results	3-77
Code Generation Pane: AUTOSAR Code Generation	
Options	3-79
Code Generation: AUTOSAR Code Generation Options Tab	
Overview	3-79
Generate XML file from schema version	3-79
Maximum SHORT-NAME length	3-81
Use AUTOSAR compiler abstraction macros	3-81
Support root-level matrix I/O using one-dimensional arrays	3-82
Code Generation: Coder Target Pane	3-83
Code Generation: Coder Target Pane Overview (previously	
“IDE Link Tab Overview”)	3-84
Coder Target: Tool Chain Automation Tab Overview	3-84
Build format	3-86
Build action	3-87
Overrun notification	3-89
Function name	3-91
Configuration	3-91
Compiler options string	3-93
Linker options string	3-94
System stack size (MAUs)	3-95
System heap size (MAUs)	3-96
Profile real-time execution	3-97
Profile by	3-99
Number of profiling samples to collect	3-100
Maximum time allowed to build project (s)	3-101
Maximum time allowed to complete IDE operation (s)	3-102
Export IDE link handle to base workspace	3-103
IDE link handle name	3-104
Source file replacement	3-105
Code Generation: Target Hardware Resources Pane	3-107
Code Generation: Coder Target Pane Overview	3-108

(Target Hardware Resources)	3-108
Coder Target: Target Hardware Resources Tab Overview	3-108
IDE/Tool Chain	3-109
Target Hardware Resources: Board Tab	3-110
Target Hardware Resources: Memory Tab	3-113
Target Hardware Resources: Section Tab	3-115
Target Hardware Resources: DSP/BIOS Tab	3-118
Target Hardware Resources: Peripherals Tab	3-121
Clocking	3-123
ADC	3-126
COMP	3-128
eCAN_A, eCAN_B	3-129
eCAP	3-131
ePWM	3-133
I2C	3-135
SCI_A, SCI_B, SCI_C	3-141
SPI_A, SPI_B, SPI_C, SPI_D	3-144
eQEP	3-147
Watchdog	3-149
GPIO	3-151
Flash_loader	3-156
DMA_ch[#]	3-158
LIN	3-167
Add Processor Dialog Box	3-174
Target Hardware Resources Tab: Linux, VxWorks, or Windows	3-175
Coder Target Pane: Altera Cyclone V SoC development kit, Arrow SoCKit development board	3-177
Coder Target Pane Overview	3-177
Coder Target	3-178
Clocking	3-178
RTOS	3-179
Build Options	3-179
External mode	3-180
Coder Target Pane: ARM Cortex-A9 (QEMU)	3-181
Coder Target Pane Overview	3-181
Coder Target	3-182
Clocking	3-182
Linux	3-183
Build Options	3-183
External mode	3-184

Coder Target Pane: ARM Cortex-M3 (QEMU)	3-185
Coder Target Pane Overview	3-185
Coder Target	3-185
Clocking	3-186
External mode	3-184
Coder Target Pane: Embedded Coder Support Package for Freescale FRDM-KL25Z Board	3-188
Coder Target Pane Overview	3-188
Coder Target	3-189
Scheduler options	3-189
Build Options	3-189
Clocking	3-190
UART0, UART1, and UART2	3-191
Coder Target Pane: BeagleBone Black Hardware	3-193
Coder Target Pane Overview	3-194
Coder Target	3-194
Board Parameters	3-195
Build Options	3-183
Clocking	3-182
Linux	3-183
External mode	3-184
Coder Target Pane: Support Package for STMicroelectronics STM32F4 Discovery Hardware	3-198
Coder Target Pane: STM32F4–Discovery Overview	3-199
Support Package for STMicroelectronics STM32F4 Discovery Hardware Settings	3-199
Scheduler options	3-200
Build options	3-201
Clocking	3-201
PIL	3-202
ADC Common	3-203
ADC 1, ADC 2, ADC 3	3-205
GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I	3-206
Coder Target Pane	3-207
System target file	3-208
Target hardware	3-209
Toolchain	3-209
Coder Target Pane: Texas Instruments C2000 Processors	3-210
Coder Target Pane: TI C2000 Processors Overview	3-211

Texas Instruments C2000 Settings	3-212
Scheduler options	3-213
Build options	3-214
Clocking	3-123
ADC	3-126
COMP	3-128
eCAN_A, eCAN_B	3-129
eCAP	3-131
ePWM	3-133
I2C	3-135
SCI_A, SCI_B, SCI_C	3-141
SPI_A, SPI_B, SPI_C, SPI_D	3-144
eQEP	3-147
Watchdog	3-149
GPIO	3-151
Flash_loader	3-156
DMA_ch[#]	3-158
LIN	3-167
Coder Target Pane	3-267
System target file	3-268
Target hardware	3-269
Toolchain	3-269
Coder Target Pane: Texas Instruments Concerto F28M3x (ARM Cortex-M3)	3-270
Coder Target Pane Overview	3-270
Coder Target	3-270
Scheduler options	3-271
Build options	3-271
Clocking	3-273
GPIO A-D	3-275
Coder Target Pane: Xilinx Zynq ZC702/ZC706 Evaluation Kits, ZedBoard	3-277
Coder Target Pane Overview	3-277
Coder Target	3-277
Clocking	3-278
Linux	3-278
Build Options	3-278
Parameter Reference	3-281
Recommended Settings Summary	3-281
Parameter Command-Line Information Summary	3-293

Embedded Coder Checks 4-2

- Embedded Coder Checks Overview 4-2
- Check for blocks not recommended for C/C++ production code deployment 4-2
- Identify lookup table blocks that generate expensive out-of-range checking code 4-3
- Check output types of logic blocks 4-5
- Check the hardware implementation 4-6
- Identify questionable software environment specifications 4-7
- Identify questionable code instrumentation (data I/O) 4-9
- Check for blocks not recommended for MISRA-C:2004 compliance 4-10
- Check configuration parameters for MISRA-C:2004 compliance 4-11
- Identify questionable subsystem settings 4-13
- Identify blocks that generate expensive fixed-point and saturation code 4-13
- Identify questionable fixed-point operations 4-16
- Identify blocks that generate expensive rounding code 4-19

Alphabetical List

activate

Mark file, project, or build configuration as active

Syntax

```
activate(IDE_Obj, 'objectname', 'type')
```

IDEs

This function supports the following IDEs:

- Analog Devices™ VisualDSP++®
- Texas Instruments™ Code Composer Studio™ v3

Description

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

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

Input Arguments

IDE_Obj

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

objectname

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

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

For build configurations, enter 'Debug', 'Release', or 'Custom'. Before using the `activate` method on a build configuration, activate the project that contains the build configuration. For more information about configurations, see “Configuration” on page 3-91.

type

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

- 'project' — Makes a specified project active.
- 'buildcfg' — Make a specified build configuration active

IDE support for type

	CCS	VisualDSP++
'project'	Yes	Yes
'buildcfg'	Yes	Yes

Examples

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

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

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

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

See Also

`build` | `new` | `remove`

activateConfigSet

Class: `cgv.CGV`

Package: `cgv`

Activate configuration set of model

Syntax

```
cgvObj.activateConfigSet(configSetName)
```

Description

`cgvObj.activateConfigSet(configSetName)` specifies the active configuration set for the model, only while the model is executed by `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. `configSetName` is the name of a configuration set object, “Simulink.ConfigSet”, which already exists in the model. The original configuration set for the model is restored after execution of the `cgv.CGV` object.

Examples

Before calling `cgv.CGV.run` on a `cgv.CGV` object for a model, the model must already contain the named configuration set. After creating the `cgv.CGV` object for a model, you can use `cgv.CGV.activateConfigSet` to activate a configuration set in the model when the `cgv.CGV` object simulates the model.

```
configObj = Simulink.ConfigSet;  
attachConfigSet('rtwdemo_cgv', configObj);  
cgvObj = cgv.CGV('rtwdemo_cgv');  
cgvObj.activateConfigSet(configObj.Name);
```

How To

- “About Model Configurations”
- “Programmatic Code Generation Verification”

add

Add files to active project in IDE

Syntax

```
add(IDE_Obj, filename, filetype)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

Before using `add`:

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

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

Supported File Types and Extensions

File Type	Extensions Supported	CCS IDE Project Folder
C/C++ source files	.c, .cpp, .cc, .cxx, .sa, .h, .hpp, .hxx	Source

File Type	Extensions Supported	CCS IDE Project Folder
Assembly source files	.a*, .s* (excluding .sa), .dsp	Source
Object and library files	.o*, .lib, .doj, .dlb	Libraries
Linker command file	.cmd, .ldf	Project Name
VDK support file	.vdk	Not applicable
DSP/BIOS file (only with CCS IDE)	.tcf	DSP/BIOS Config

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

Input Arguments

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

IDE_Obj

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

filename

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

If you supply a filename without a path or relative path, your coder product searches the IDE working folder first. It then searches the folders on your MATLAB path. Add supported file types shown in the preceding table.

filetype

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

Examples

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

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

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

See Also

`activate` | `"cd"` | `open` | `remove` | `new`

addAdditionalHeaderFile

Add header file to array of header files for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

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 code replacement table entry.

Examples

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 code replacement table entry.

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

op_entry = RTW.Tf1COperationEntry;
.
.
.
```

```
addAdditionalHeaderFile(op_entry, 'all_additions.h');  
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));  
  
addAdditionalSourceFile(op_entry, 'all_additions.c');  
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

More About

- “Specify Build Information for Replacement Code”
- “Define Code Replacement Mappings”

See Also

[addAdditionalIncludePath](#) | [addAdditionalSourceFile](#) |
[addAdditionalSourcePath](#)

addAdditionalIncludePath

Add include path to array of include paths for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

path

String specifying the full path to an additional header file. The string can include tokens (for example, *\$myfolder\$*, where *myfolder* is a variable defined as a string or cell array of strings in the MATLAB workspace).

Description

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

Examples

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 code replacement table entry.

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

op_entry = RTW.Tf1COperationEntry;
```

```
.  
. .  
addAdditionalHeaderFile(op_entry, 'all_additions.h');  
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));  
  
addAdditionalSourceFile(op_entry, 'all_additions.c');  
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

More About

- “Specify Build Information for Replacement Code”
- “Define Code Replacement Mappings”

See Also

`addAdditionalHeaderFile` | `addAdditionalSourceFile` |
`addAdditionalSourcePath`

addAdditionalLinkObj

Add link object to array of link objects for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

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 code replacement table entry.

Examples

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

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

op_entry = RTW.Tf1COperationEntry;
...
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
```


More About

- “Specify Build Information for Replacement Code”
- “Define Code Replacement Mappings”

See Also

`addAdditionalLinkObjPath`

addAdditionalLinkObjPath

Add link object path to array of link object paths for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

path

String specifying the full path to an additional link object. The string can include tokens (for example, *\$myfolder\$*, where *myfolder* is a variable defined as a string or cell array of strings in the MATLAB workspace).

Description

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

Examples

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

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

op_entry = RTW.Tf1COperationEntry;
...
```

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

More About

- “Specify Build Information for Replacement Code”
- “Define Code Replacement Mappings”

See Also

addAdditionalLinkObj

addAdditionalSourceFile

Add source file to array of source files for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

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 code replacement table entry.

Examples

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 code replacement table entry.

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

op_entry = RTW.Tf1COperationEntry;
.
.
.
```

```
addAdditionalHeaderFile(op_entry, 'all_additions.h');  
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));  
  
addAdditionalSourceFile(op_entry, 'all_additions.c');  
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

More About

- “Specify Build Information for Replacement Code”
- “Define Code Replacement Mappings”

See Also

[addAdditionalHeaderFile](#) | [addAdditionalIncludePath](#) |
[addAdditionalSourcePath](#)

addAdditionalSourcePath

Add source path to array of source paths for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

path

String specifying the full path to an additional source file. The string can include tokens (for example, *\$myfolder\$*, where *myfolder* is a variable defined as a string or cell array of strings in the MATLAB workspace).

Description

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

Examples

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 code replacement table entry.

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

op_entry = RTW.Tf1COperationEntry;
```

```
.  
. .  
addAdditionalHeaderFile(op_entry, 'all_additions.h');  
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));  
  
addAdditionalSourceFile(op_entry, 'all_additions.c');  
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
```

More About

- “Specify Build Information for Replacement Code”
- “Define Code Replacement Mappings”

See Also

[addAdditionalHeaderFile](#) | [addAdditionalIncludePath](#) | [addAdditionalSourceFile](#)

addAlgorithmProperty

Add algorithm properties for code replacement table entry

Syntax

```
addAlgorithmProperty(hEntry, name-value)
```

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

name-value

Algorithm property, specified as a comma-separated pair consisting of the name of an algorithm property and one or more algorithm values. Specify multiple values as a cell array of strings.

Name	Values
'ExtrapMethod'	'Clip' 'Linear'
'IndexSearchMethod'	'Evenly spaced points' 'Linear search' 'Binary search'
'InterpMethod'	'Flat' 'Linear' 'Cubic spline'
'IndexSearchMethod'	'Evenly spaced points' 'Linear search' 'Binary search'
'NumberOfTableDimensions'	'1' '2' '3' '4' '5'
'RemoveProectionInput'	'off' 'on'
'RndMeth'	'Ceiling' 'Convergent' 'Floor' 'Nearest' 'Round' 'Simplest' 'Zero'
'SaturateOnIntegerOverflow'	'off' 'on'
'UseLastBreakpoint'	'off' 'on'

Name	Values
'UseLastTableValue'	'off' 'on'
'ValidIndexMayReachLast'	'off' 'on'

Description

The `addAlgorithmProperty` function adds algorithm property settings to the conceptual representation of a code replacement table entry. For example, use this function to adjust the algorithms applied by lookup table functions.

Examples

In the following example, the `addAlgorithmProperty` function configures the code generator to apply the following methods when replacing code for the `lookup1D` function:

- Clip extrapolation
- Linear interpolation
- Binary or linear index search

```

hLib = RTW.Tf1Table;

hEnt = RTW.Tf1CFunctionEntry;
hEnt.setTf1CFunctionEntryParameters( ...
    'Key', 'lookup1D', ...
    'Priority', 100, ...
    'ImplementationName', 'my_Lookup1D_Repl', ...
    'ImplementationHeaderFile', 'my_Lookup1D.h', ...
    'ImplementationSourceFile', 'my_Lookup1D.c', ...
    'GenCallback', 'RTW.copyFileToBuildDir');

arg = hEnt.getTf1ArgFromString('y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
hEnt.addConceptualArg(arg);

arg = hEnt.getTf1ArgFromString('u1','double');
hEnt.addConceptualArg(arg);

arg = RTW.Tf1ArgMatrix('u2','RTW_IO_INPUT','double');
arg.DimRange = [0 0; Inf Inf];
hEnt.AddConceptualArg(arg);

arg = RTW.Tf1ArgMatrix('u3', 'RTW_IO_INPUT', 'double');
arg.DimRange = [0 0; Inf Inf];

```

```
hEnt.addConceptualArg(arg);  
hEnt.addAlgorithmProperty('ExtrapMethod', 'Clip');  
hEnt.addAlgorithmProperty('InterpMethod', 'Linear');  
hEnt.addAlgorithmProperty('IndexSearchMethod', {'Linear search', ...  
                                                'Binary search'});
```

More About

- “Lookup Table Function Code Replacement”

See Also

getTflArgFromString

addArgConf

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

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

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

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

Input Arguments

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

<i>category</i>	String specifying the argument category, either 'Value' or 'Pointer'.
<i>argName</i>	String specifying a valid C identifier.
<i>qualifier</i>	String specifying the argument type qualifier: 'none', 'const', 'const *', or 'const * const'.

Examples

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

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

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

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

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

Alternatives

You can specify the argument configuration information in the Model Interface dialog box. See “Configure Function Prototypes Using Graphical Interfaces” in the Embedded Coder[®] documentation.

See Also

`RTW.ModelSpecificCPrototype.attachToModel`

How To

- “Function Prototype Control”

addBaseline

Class: `cgv.CGV`

Package: `cgv`

Add baseline file for comparison

Syntax

```
cgvObj.addBaseline(inputName,baselineFile)
```

```
cgvObj.addBaseline(inputName,baselineFile,toleranceFile)
```

Description

cgvObj.addBaseline(inputName,baselineFile) associates a baseline data file to an inputName in *cgvObj*. *cgvObj* is a handle to a `cgv.CGV` object. If a baseline file is present, when you call `cgv.CGV.run`, *cgvObj* automatically compares baseline data to the result data of the current execution of *cgvObj*.

cgvObj.addBaseline(inputName,baselineFile,toleranceFile) includes an optional tolerance file to apply when comparing the baseline data to the result data of the current execution of *cgvObj*.

Input Arguments

inputName

A unique numeric or character identifier assigned to the input data associated with baselineFile

baselineFile

A MAT-file containing baseline data

toleranceFile

File containing the tolerance specification, which is created using `cgv.CGV.createToleranceFile`

Examples

A typical workflow for defining baseline data in a `cgv.CGV` object and then comparing the baseline data to the execution data is as follows:

- 1 Create a `cgv.CGV` object for a model.
- 2 Add input data to the `cgv.CGV` object by calling `cgv.CGV.addInputData`.
- 3 Add the baseline file to the `cgv.CGV` object by calling `cgv.CGV.addBaseline`, which associates the `inputName` for input data in the `cgv.CGV` object with input data stored in the `cgv.CGV` object as the baseline data.
- 4 Run the `cgv.CGV` object by calling `cgv.CGV.run`, which automatically compares the baseline data to the result data in this execution.
- 5 Call `cgv.CGV.getStatus` to determine the results of the comparison.

See Also

`cgv.CGV.addInputData` | `cgv.CGV.getStatus` | `cgv.CGV.run` | `cgv.CGV.createToleranceFile`

How To

- “Verify Numerical Equivalence with CGV”

addHeaderReportFcn

Class: `cgv.CGV`

Package: `cgv`

Add callback function to execute before executing input data in object

Syntax

```
cgvObj.addHeaderReportFcn(CallbackFcn)
```

Description

`cgvObj.addHeaderReportFcn(CallbackFcn)` adds a callback function to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. “`run (cgv.CGV)`” calls `CallbackFcn` before executing input data included in `cgvObj`. The callback function signature is:

```
CallbackFcn(cgvObj)
```

Examples

The callback function, `HeaderReportFcn`, is added to `cgv.CGV` object, `cgvObj`

```
cgvObj.addHeaderReportFcn(@HeaderReportFcn);
```

where `HeaderReportFcn` is defined as:

```
function HeaderReportFcn(cgvObj)
...
end
```

See Also

`cgv.CGV.run`

How To

- “Callbacks for Customized Model Behavior”

addPostExecFcn

Class: `cgv.CGV`

Package: `cgv`

Add callback function to execute after each input data file is executed

Syntax

```
cgvObj.addPostExecFcn(CallbackFcn)
```

Description

`cgvObj.addPostExecFcn(CallbackFcn)` adds a callback function to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. “`run (cgv.CGV)`” calls *CallbackFcn* after each input data file is executed for the model. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numerical identifier associated with input data in the `cgvObj`.

Examples

The callback function, *PostExecutionFcn*, is added to `cgv.CGV` object, `cgvObj`

```
cgvObj.addPostExecFcn(@PostExecutionFcn);
```

where *PostExecutionFcn* is defined as:

```
function PostExecutionFcn(cgvObj, inputIndex)
...
end
```

See Also

`cgv.CGV.run`

How To

- “Callbacks for Customized Model Behavior”

addPostExecReportFcn

Class: `cgv.CGV`

Package: `cgv`

Add callback function to execute after each input data file executes

Syntax

```
cgvObj.addPostExecReportFcn(CallbackFcn)
```

Description

`cgvObj.addPostExecReportFcn(CallbackFcn)` adds a callback function to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. “`run (cgv.CGV)`” calls *CallbackFcn* after each input data file is executed for the model. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numeric identifier associated with input data in the `cgvObj`.

Examples

The callback function, *PostExecutionReportFcn*, is added to `cgv.CGV` object, `cgvObj`

```
cgvObj.addPostExecReportFcn(@PostExecutionReportFcn);
```

where *PostExecutionReportFcn* is defined as:

```
function PostExecutionReportFcn(cgvObj, inputIndex)
...
end
```

See Also

`cgv.CGV.run`

How To

- “Callbacks for Customized Model Behavior”

addPreExecFcn

Class: `cgv.CGV`

Package: `cgv`

Add callback function to execute before each input data file executes

Syntax

```
cgvObj.addPreExecFcn(CallbackFcn)
```

Description

`cgvObj.addPreExecFcn(CallbackFcn)` adds a callback function to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. “`run (cgv.CGV)`” calls `CallbackFcn` before executing each input data file in `cgvObj`. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

`inputIndex` is a unique numeric identifier associated with input data in `cgvObj`.

Examples

The callback function, `PreExecutionFcn`, is added to `cgv.CGV` object, `cgvObj`

```
cgvObj.addPreExecFcn(@PreExecutionFcn);
```

where `PreExecutionFcn` is defined as:

```
function PreExecutionFcn(cgvObj, inputIndex)
...
end
```

See Also

`cgv.CGV.run`

How To

- “Callbacks for Customized Model Behavior”

addPreExecReportFcn

Class: `cgv.CGV`

Package: `cgv`

Add callback function to execute before each input data file executes

Syntax

```
cgvObj.addPreExecReportFcn(CallbackFcn)
```

Description

`cgvObj.addPreExecReportFcn(CallbackFcn)` adds a callback function to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. “`run (cgv.CGV)`” calls *CallbackFcn* before executing each input data file in `cgvObj`. The callback function signature is:

```
CallbackFcn(cgvObj, inputIndex)
```

inputIndex is a unique numerical identifier associated with input data in `cgvObj`.

Examples

The callback function, *PreExecutionReportFcn*, is added to `cgv.CGV` object, `cgvObj`

```
cgvObj.addPreExecReportFcn(@PreExecutionReportFcn);
```

where *PreExecutionReportFcn* is defined as:

```
function PreExecutionReportFcn(cgvObj, inputIndex)
...
end
```

See Also

`cgv.CGV.run`

How To

- “Callbacks for Customized Model Behavior”

addTrailerReportFcn

Class: `cgv.CGV`

Package: `cgv`

Add callback function to execute after the input data executes

Syntax

```
cgvObj.addTrailerReportFcn(CallbackFcn)
```

Description

`cgvObj.addTrailerReportFcn(CallbackFcn)` adds a callback function to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. “`run (cgv.CGV)`” executes the input data files in `cgvObj` and then calls `CallbackFcn`. The callback function signature is:

```
CallbackFcn(cgvObj)
```

Examples

The callback function, `TrailerReportFcn`, is added to `cgv.CGV` object, `cgvObj`

```
cgvObj.addTrailerReportFcn(@TrailerReportFcn);
```

where `TrailerReportFcn` is defined as:

```
function TrailerReportFcn(cgvObj)
...
end
```

See Also

`cgv.CGV.run`

How To

- “Callbacks for Customized Model Behavior”

addCheck

Class: `rtw.codegenObjectives.Objective`

Package: `rtw.codegenObjectives`

Add checks

Syntax

```
addCheck(obj, checkID)
```

Description

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

Input Arguments

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

Examples

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

```
addCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

See Also

`Simulink.ModelAdvisor`

How To

- “Create Custom Objectives”

- “About IDs”

addComplexTypeAlignment

Specify alignment boundary of a complex type

Syntax

```
addComplexTypeAlignment(hDataAlign, baseType, alignment)
```

Arguments

hDataAlign

Handle to a data alignment object, previously returned by *hDataAlign* = RTW.DataAlignment.

baseType

String specifying a built-in data type such as `int8` or `long`.

alignment

A positive integer that is a power of 2 and does not exceed 128. This value specifies the alignment boundary.

Description

The `addComplexTypeAlignment` function specifies the alignment boundary of real and complex data members of a complex type. The starting memory address of the real and imaginary part of complex variables produced by the code generator with the specified type are a multiple of the specified alignment boundary. The code generator replaces operations in generated code if a code replacement table entry has a complex argument with a data alignment requirement that is less than or equal to the alignment boundary value and the entry satisfies all other code replacement match criteria.

To use this function, your code replacement library registration file must include additional compiler data alignment information, such as alignment syntax.

Examples

Specify a 16-byte alignment boundary for complex `int8` types by adding the following lines of code to your code replacement library registration file.

```
da = RTW.DataAlignment;  
addComplexTypeAlignment(da, 'int8', 16);
```

More About

- “Provide Data Alignment Specifications for Compilers”
- “Data Alignment for Code Replacement”
- “Define Code Replacement Mappings”

addConceptualArg

Add conceptual argument to array of conceptual arguments for code replacement table entry

Syntax

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

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

arg

Argument, such as returned by *arg* = getTf1ArgFromString(*name*, *datatype*), to be added to the array of conceptual arguments for the code replacement table entry.

Description

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

Examples

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.Tf1Table;  
  
% Create entry for addition of built-in uint8 data type  
op_entry = RTW.Tf1COperationEntry;  
op_entry.setTf1COperationEntryParameters( ...
```

```
'Key',                'RTW_OP_ADD', ...
'Priority',           90, ...
'SaturationMode',   'RTW_SATURATE_ON_OVERFLOW', ...
'RoundingModes',    {'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 );
```

More About

- “Define Code Replacement Mappings”

See Also

[getTflArgFromString](#)

addDWorkArg

Add DWork argument for semaphore entry in code replacement table

Syntax

```
addDWorkArg(hEntry, arg)
```

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement semaphore table entry class, using *hEntry* = RTW.Tf1CSemaphoreEntry.

arg

Argument, such as returned by *arg* = getTf1DWorkFromString(*name*, *datatype*), to be added to the arguments for the code replacement table entry.

Description

The addDWorkArg function adds a specified DWork argument to the arguments for a semaphore entry in a code replacement table.

Examples

In the following example, the addDWorkArg function is used to add a DWork argument named *d1* to the arguments for a semaphore entry in a code replacement table.

```
hLib = RTW.Tf1Table;  
sem_entry = RTW.Tf1CSemaphoreEntry;  
.  
.  
.  
% DWork Arg  
  
arg = hLib.getTf1DWorkFromString('d1','void*');
```

```
sem_entry.addDWorkArg( arg );
```

```
hLib.addEntry( sem_entry );
```

More About

- “Semaphore and Mutex Function Replacement”
- “Define Code Replacement Mappings”

See Also

`getTf1DWorkFromString`

addConfigSet

Class: `cgv.CGV`

Package: `cgv`

Add configuration set

Syntax

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

Description

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

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

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

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

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

How To

- “Programmatic Code Generation Verification”
- “About Model Configurations”

addEntry

Add table entry to collection of table entries registered in code replacement table

Syntax

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

Arguments

hTable

Handle to a code replacement table previously returned by *hTable* = RTW.Tf1Table.

entry

Handle to a function or operator entry that you have constructed after calling *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry

Description

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

Examples

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

```
hLib = RTW.Tf1Table;

% Create an entry for addition of built-in uint8 data type
op_entry = RTW.Tf1COperationEntry;
op_entry.setTf1COperationEntryParameters( ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 'u8_add_u8_u8', ...
```

```
        'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
        'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTf1ArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );

arg = hLib.getTf1ArgFromString('u1','uint8');
op_entry.addConceptualArg( arg );

arg = hLib.getTf1ArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

More About

- “Define Code Replacement Mappings”

addInputData

Class: `cgv.CGV`

Package: `cgv`

Add input data

Syntax

```
cgvObj.addInputData(inputName, inputDataFile)
```

Description

`cgvObj.addInputData(inputName, inputDataFile)` adds an input data file to `cgvObj`. `cgvObj` is a handle to a `cgv.CGV` object. `inputName` is a unique identifier, which `cgvObj` associates with the input data in `inputDataFile`.

Tips

- When calling `addInputData` you can modify configuration parameters by including their settings in the input file, `inputDataFile`.
- If you omit calling `addInputData` before executing the model, the `cgv.CGV` object runs once using data in the base workspace.
- The `cgvObj` uses the `inputName` to identify the input data associated with output data and output data files. `cgvObj` passes `inputName` to a callback function to identify the input data that the callback function uses.

Input Arguments

`inputName`

`inputName` is a unique numeric or character identifier, which is associated with the input data in `inputDataFile`.

inputDataFile

`inputDataFile` is an input data file, with or without the `.mat` extension. `cgvObj` uses the input data when the model executes during `cgv.CGV.run`. If the input file is in the working folder, the `cgvObj` does not require the path. `addInputData` does not qualify that the contents of `inputDataFile` relate to the inputs of the model. Data that is not used by the model will not throw a warning or error.

See Also

`cgv.CGV.run`

How To

- “Verify Numerical Equivalence with CGV”

addParam

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Add parameters

Syntax

```
addParam(obj, paramName, value)
```

Description

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

Input Arguments

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

Examples

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

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

See Also

`get_param`

How To

- “Create Custom Objectives”

- “Parameter Command-Line Information Summary”

addPostLoadFiles

Class: `cgv.CGV`

Package: `cgv`

Add files required by model

Syntax

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

Description

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

Note: Subsequent `cgvObj.addPostLoadFiles` calls to the same `cgv.CGV` object replaces the list of MATLAB and MAT-files of that object.

How To

- “Verify Numerical Equivalence with CGV”
- “Callbacks for Customized Model Behavior”

address

Memory address and page value of symbol in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

Input Arguments

a

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

IDE_Obj

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

symbol

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

Symbol names are case sensitive.

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

scope

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

Output Arguments

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

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

For Code Composer Studio

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

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

For VisualDSP++

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

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

Examples

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

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

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

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

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

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

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

MATLAB software returns

```
ddatv =  
    3.1416    12.3    0.3679    0.7071
```

See Also

`load` | `read` | “symbol” | `write`

adivdsp

Create handle object to interact with VisualDSP++ IDE

Syntax

```
IDE_Obj = adivdsp
IDE_Obj = adivdsp('proprname1',propvalue1,'proprname2',propvalue2,...
,'timeout',value)
IDE_Obj = adivdsp('my_session')
```

Note: The output object name (left side argument) you provide for `adivdsp` cannot begin with an underscore, such as `_IDE_Obj`.

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++

Description

If the IDE is not running, `IDE_Obj = adivdsp` opens the VisualDSP++ software for the most recent active session. After that, it creates an object, `IDE_Obj`, that references the newly opened session. If the IDE is running, `adivdsp` returns object `IDE_Obj` that connects to the active session in the IDE.

`adivdsp` creates an interface between MATLAB software and Analog Devices VisualDSP++ software. The first time you use `adivdsp`, supply a session name as an input argument (refer to the next syntax).

`IDE_Obj = adivdsp('sessionname','name','procnum','number',...)` returns an object handle `IDE_Obj` that you use to interact with a processor in the IDE from MATLAB.

Use the debug methods with this object to access memory and control the execution of the processor.

The `adivdsp` function interprets input arguments as object property definitions. Each property definition consists of a property name followed by the desired property value (often called a *PV*, or *property name/property value*, pair). Although you can define a number of `adivdsp` object properties when you create the object, there are several important properties that you must provide during object construction. These properties must be delineated when you create the object. The required input arguments are as follows:

- **sessionname** — Specifies the session to connect to. This session must exist in the session list. `adivdsp` does not create new sessions. The resulting object refers to a processor in **sessionname**. To see the list of sessions, use `listsessions` at the MATLAB command prompt.
- **procnum**— Specifies the processor to connect to in **sessionname**. The `adivdsp` object only supports connecting to processor 0. As such, the default value for **procnum** is 0 for the first processor on the board. If you omit the **procnum** argument, `adivdsp` connects to the first processor.

After you build the `adivdsp` object `IDE_Obj`, you can review the object property values with `get`, but you cannot modify the **sessionname** and **procnum** property values.

To connect to the active session in IDE, omit the **sessionname** property in the syntax. If you do not pass **sessionname** as an input argument, the object defaults to the active session in the IDE.

Use `listsessions` to determine the number for the desired DSP processor. If your IDE session is single processor or to connect to processor zero, you can omit the **procnum** property definition. If you omit the *procnum* argument, *procnum* defaults to 0 (zero-based).

`IDE_Obj = adivdsp('propname1',propvalue1,'propname2',propvalue2,..., 'timeout', value)` sets the global time-out value to `value` in `IDE_Obj`. MATLAB waits for the specified time-out value to get a response from the IDE application. If the IDE does not respond within the allotted time-out period, MATLAB exits from the evaluation of this function.

If the session exists in the session list and the IDE is not already running, `IDE_Obj = adivdsp('my_session')` connects to `my_session`. In this case, MATLAB starts VisualDSP++ IDE for the session named `my_session`.

The following list shows some other possible cases and results of using `adivdsp` to construct an object that refers to `my_session`.

- If `my_session` does not exist in the session list and the IDE is not already running, MATLAB returns an error stating that `my_session` does not exist in the session list.
- When `my_session` is the current active session and the IDE is already running, MATLAB connects to the IDE for this session.
- If `my_session` is not the current active session, but exists in the session list, and the IDE is already running, MATLAB displays a dialog box asking if you want to switch to `my_session`. If you choose to switch to `my_session`, the existing handles you have to other sessions in the IDE become invalid. To connect to the other sessions you use `adivdsp` to recreate the objects for those sessions.
- If `my_session` does not exist in the session list and the IDE is already running, MATLAB returns an error, explaining that the session `my_session` does not exist in the session list.

Examples

These examples show some of the operation of `adivdsp`.

```
IDE_Obj = adivdsp('sessionname', 'my_session', 'procnum', 0);
```

returns a handle to the first DSP processor for session `my_session`.

`IDE_Obj = adivdsp` without input arguments constructs the object `IDE_Obj` with the default property values, returning a handle to the first DSP processor for the active session in the IDE.

```
IDE_Obj = adivdsp('sessionname', 'my_session');
```

returns a handle to the first DSP processor for the session `my_session`.

See Also

`listsessions`

adivdspsetup

Configure your coder product to interact with VisualDSP++ IDE

Syntax

```
adivdspsetup
```

IDEs

This function supports the following IDEs:

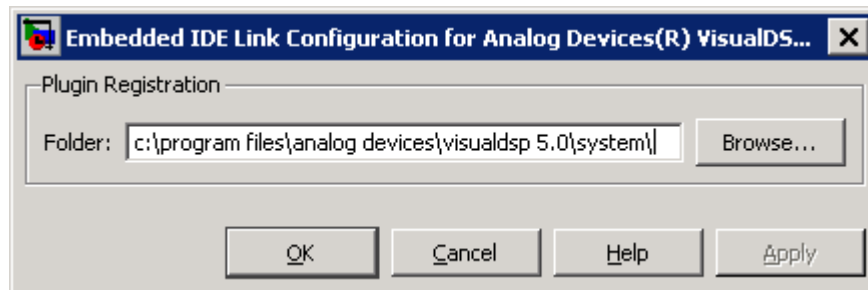
- Analog Devices VisualDSP++

Description

Enter `adivdspsetup` at the MATLAB command line when you are setting up your coder product to interact with VisualDSP++ for the first time. This action displays a dialog box to specify where to install a plug-in for VisualDSP++. The default value for **Folder** is the VisualDSP++ `system` folder. You can specify folders for which you have write access. When you click **OK**, the software adds the plug-in to the folder and registers the plug-in with the VisualDSP++ IDE.

Examples

- 1 At the MATLAB command line, enter: `adivdspsetup`. This action opens the following dialog box:



- 2 Click **Browse**, locate the `system` folder for VisualDSP++, and click **OK**. This action registers the MathWorks plugin to the VisualDSP++ IDE.

See Also

adivdsp

animate

Run application on processor to breakpoint

Syntax

```
animate(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`animate(IDE_Obj)` starts the processor application, which runs until it encounters a breakpoint in the code. At the breakpoint, application execution halts and CCS Debugger returns data to the IDE to update the windows not connected to probe points. After updating the display, the application resumes execution and runs until it encounters another breakpoint. The run-break-resume process continues until you stop the application from MATLAB software with the `halt` function or from the IDE.

While running scripts or files in MATLAB software, you can use `animate` to update the IDE with information as your script or program runs.

Using animate with Multiprocessor Boards

When you use `animate` with a `ticcs` object `IDE_Obj` that comprises more than one processor, such as an OMAP processor, the method applies to each processor in your `IDE_Obj` object. This action causes each processor to run a loaded program just as it does for the single processor case.

See Also

`halt` | `restart` | `run`

attachToModel

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

attachToModel

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Alternatives

Click the **Configure Model Functions** button on the **Code Generation > Interface** pane of the Configuration Parameters dialog box for flexible control over the model function prototypes that are generated for your model. Once you validate and apply your changes, you can generate code based on your function prototype modifications. See “Configure Function Prototypes Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

build

Build or rebuild current project

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

See Also

isrunning | open

ccsboardinfo

Information about boards and simulators known to IDE

Syntax

```
ccsboardinfo
boards = ccsboardinfo
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`ccsboardinfo` returns configuration information about each board and processor installed and recognized by CCS. When you issue the function, `ccsboardinfo` returns the following information about each board or simulator.

Installed Board Configuration Data	Configuration Item Name	Description
Board number	boardnum	The number CCS assigns to the board or simulator. Board numbering starts at 0 for the first board. You also use <code>boardnum</code> when you create a link to the IDE.
Board name	boardname	The name assigned to the board or simulator. Usually, the name is the board model name, such as TMS320C67xx evaluation module. If you are using a simulator, the name tells you which processor the simulator matches, such as C67xx simulator. If you renamed the board during setup, this item displays the board name.

Installed Board Configuration Data	Configuration Item Name	Description
Processor number	<code>procnum</code>	The number assigned by CCS to the processor on the board or simulator. When the board contains more than one processor, CCS assigns a number to each processor, numbering from 0 for the first processor on the first board. For example, when you have two boards, the first processor on the first board is <code>procnum = 0</code> , and the first and second processors on the second board are <code>procnum = 1</code> and <code>procnum = 2</code> . You also use this property when you create a link to the IDE.
Processor name	<code>procname</code>	Provides the name of the processor. Usually the name is CPU, unless you assign a different name.
Processor type	<code>proctype</code>	Gives the processor model, such as TMS320C6x1x for the C6xxx series processors.

Each row in the table that you see displayed represents one digital signal processor, either on a board or simulator. As a consequence, you use the information in the table in the function `ticcs` to identify a selected board in your PC.

`boards = ccsboardinfo` returns the configuration information about your installed boards in a slightly different manner. Rather than return the table of the information, the method returns a list of board names and numbers. In that list, each board has a structure named `proc` that contains processor information. For example

```
boards = ccsboardinfo
```

```
returns
```

```
boards =
```

```
    name: 'C6xxx Simulator (Texas Instruments)'  
    number: 0  
    proc: [1x1 struct]
```

where the structure `proc` contains the processor information for the C6xxx simulator board:

```
boards.proc
```

```
ans =
    name: 'CPU'
  number: 0
   type: 'TMS320C6200'
```

Reviewing the output from both function syntaxes shows that the configuration information is the same.

To connect with a specific board when you create an IDE handle object, combine this syntax with the dot notation for accessing elements in a structure. Use the `boardnum` and `procnum` properties in the `boards` structure. For example, when you enter

```
boards = ccsboardinfo;
```

`boards(1).name` returns the name of your second installed board and `boards(1).proc(2).name` returns the name of the second processor on the second board. To create a link to the second processor on the second board, use

```
IDE_Obj = ticcs('boardnum',boards(1).number,'procnum',...
boards(1).proc(2).name);
```

Examples

On a PC with both a simulator and a DSP Starter Kit (DSK) board installed,

```
ccsboardinfo
```

returns something like the following table. Your display may differ slightly based on what you called your boards when you configured them in CCS Setup Utility:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum ..	0	CPU	TMS320C6200
0	DSK (Texas Instruments)	0	CPU_3	TMS320C6x1x

When you have one or more boards that have multiple CPUs, `ccsboardinfo` returns the following table, or one like it:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	C6xxx Simulator (Texas Instrum .0	0	CPU	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	1	CPU_Primary	TMS320C6200

```
1 C6xxx EVM (Texas Instrum ... 0 CPU_Secondary TMS320C6200
0 C64xx Simulator (Texas Instru...0 CPU TMS320C64xx
```

In this example, board number 1 returns two defined CPUs: `CPU_Primary` and `CPU_Secondary`. The C6xxx does not in fact have two CPUs; a second CPU is defined for this example.

To show the `boards = ccsboardinfo` syntax, this example assumes a PC with two boards installed, one of which has three CPUs.

Enter the following command:

```
ccsboardinfo
```

This command generates a list of boards. For example:

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum	.0	CPU	TMS320C6211
0	C62xx DSK (Texas Instruments)	2	CPU_3	TMS320C6x1x
0	C62xx DSK (Texas Instruments)	1	CPU_4_1	TMS320C6x1x
0	C62xx DSK (Texas Instruments)	0	CPU_4_2	TMS320C6x1x

Now enter

```
boards = ccsboardinfo
```

MATLAB software returns

```
boards =
2x1 struct array with fields
    name
    number
    proc
```

showing that you have two boards in your PC.

Use the dot notation to determine the names of the boards:

```
boards.name
```

returns

```
ans =
C6xxx Simulator (Texas Instruments)
```



```
ans =  
C62xx DSK (Texas Instruments)
```

To identify the processors on each board, again use the dot notation to access the processor information. You have two boards (numbered 0 and 1). Board 0 has three CPUs defined for it. To determine the type of the second processor on board 0 (the board whose `boardnum = 0`), enter

```
boards(2).proc(1)
```

which returns

```
ans =  
    name: 'CPU_3'  
    number: 1  
    type: 'TMS320C6x1x'
```

Recall that

```
boards(2).proc
```

gives you this information about the board

```
ans =  
3x1 struct array with fields:  
    name  
    number  
    type
```

indicating that this board has three processors (the `3x1` array).

The dot notation is useful for accessing the contents of a structure when you create a link to the IDE. When you use `ticcs` to create your CCS link, you can use the dot notation to tell the IDE which processor you are using.

```
IDE_Obj = ticcs('boardnum',boards(1).proc(1))
```

See Also

`info | ticcs`

cd

Set working folder in IDE

Syntax

```
wd = cd(IDE_Obj)  
cd(IDE_Obj,folder)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

`wd = cd(IDE_Obj)` assigns the IDE working folder to the variable, `wd`, which you reference via the IDE handle object, `IDE_Obj`.

`cd(IDE_Obj,folder)` sets the IDE working folder to '`folder`'. '`folder`' can be a path string relative to your working folder, or an absolute path. The intended folder must exist. `cd` does not create a folder. Setting the IDE folder does not change your MATLAB Current Folder.

`cd` alters the default folder for `open` and `load`. Loading a new workspace file also changes the working folder for the IDE.

See Also

`dir` | `load` | `open`

cgv.CGV class

Package: cgv

Verify numerical equivalence of results

Description

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

`cgv.CGV` and `cgv.Config` use two of the same properties. Before executing a `cgv.CGV` object, use `cgv.Config` to verify the model configured for the mode of execution that you specify. If the top model is set to normal simulation mode, referenced models set to PIL mode are changed to Accelerator mode.

Construction

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

`cgvObj = cgv.CGV(model_name, Name, Value)` constructs the object using the parameter values, specified as `Name, Value` pair arguments. Parameter names and values are not case sensitive.

Input Arguments

model_name

Name of the model that you are verifying.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name, Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes

(' '). You can specify several name-value pair arguments in a variety of orders, such as Name1, Value1, ..., NameN, ValueN.

'ComponentType'

Define the SIL or PIL approach

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

If mode of execution is simulation (**Connectivity** is **sim**), choosing either value for **ComponentType** does not alter simulation results.

Default: topmodel

'Connectivity'

Specify mode of execution

Value	Description
sim or normal (default)	Mode of execution is Normal simulation.
sil	Mode of execution is SIL.
pil	Mode of execution is PIL.

Properties

Description

Specify a description of the object.

Default: ' ' (null string)

Name

Specify a name for the object.

Default: ' ' (null string)

Methods

activateConfigSet	Activate configuration set of model
addBaseline	Add baseline file for comparison
addHeaderReportFcn	Add callback function to execute before executing input data in object
addPostExecFcn	Add callback function to execute after each input data file is executes
addPostExecReportFcn	Add callback function to execute after each input data file executes
addPreExecFcn	Add callback function to execute before each input data file executes
addPreExecReportFcn	Add callback function to execute before each input data file executes
addTrailerReportFcn	Add callback function to execute after the input data executes
addConfigSet	Add configuration set
addInputData	Add input data
addPostLoadFiles	Add files required by model
compare	Compare signal data

<code>copySetup</code>	Create copy of <code>cgv.CGV</code> object
<code>createToleranceFile</code>	Create file correlating tolerance information with signal names
<code>getOutputData</code>	Get output data
<code>getSavedSignals</code>	Display list of signal names to command line
<code>getStatus</code>	Return execution status
<code>plot</code>	Create plot for signal or multiple signals
<code>run</code>	Execute <code>CGV</code> object
<code>setMode</code>	Specify mode of execution
<code>setOutputDir</code>	Specify folder
<code>setOutputFile</code>	Specify output data file name

Copy Semantics

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

Examples

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

- 1 Create a `cgv.CGV` object, `cgvObj`, for each mode of execution and use the `cgv.CGV` set up methods to configure the model for each execution. The set up methods are:

- `addInputData`
 - `addPostLoadFiles`
 - `setOutputDir`
 - `setOutputFile`
 - `addCallBack`
 - `addConfigSet`
- 2 Run the model for each mode of execution using the `cgvObj.run` method.
 - 3 Use the `cgv.CGV` access methods to get and evaluate the data. The access methods are:
 - `getOutputData`
 - `getSavedSignals`
 - `plot`
 - `compare`

An object should be run only once. After the object is run, the set up methods are not used for that object. You then use the access methods for verifying the numerical equivalence of the results.

See Also

`cgv.Config`

How To

- “Verify Numerical Equivalence with CGV”
- Using Code Generation Verification

cgv.Config class

Package: cgV

Check and modify model configuration parameter values

Description

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

To execute the model in the mode that you specify, you might need to make additional modifications to the configuration parameter values or the model beyond those configured by the `cgv.Config` object.

By default, `cgv.Config` modifies configuration parameter values to the values that it recommends, but does not save the model. Alternatively, you can use `cgv.Config` parameters to modify the default specification. For more information, see the properties, `ReportOnly` and `SaveModel`.

If you use `cgv.Config` to modify a model, do not use referenced configuration sets in that model. If a model uses a referenced configuration set, update the model with a copy of the configuration set, by using the `Simulink.ConfigSetRef.getRefConfigSet` method.

If you use `cgv.Config` on a model that executes a callback function, the callback function might modify configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. If this change occurs, the model might not be set up for SIL or PIL. For more information, see “Callbacks for Customized Model Behavior”.

Construction

`cfgObj = cgv.Config(model_name)` creates a handle to a `cgv.Config` object, `cfgObj`, using default values for properties. `model_name` is the name of the model that you are checking and optionally configuring.

`cfgObj = cgv.Config(model_name, Name, Value)` constructs the object using options, specified as parameter name and value pairs. Parameter names and values are not case sensitive.

`Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in a variety of orders, such as `Name1, Value1, ..., NameN, ValueN`.

Properties

CheckOutputs

Specify whether to compile the model and check that the model outputs configuration is compatible with the `cgv.CGV` object. If your script fixes errors reported by `cgv.Config`, you can set `CheckOutputs` to `off`.

Value	Description
on (default)	Compile the model and check the model outputs configuration
off	Do not compile the model or check the model outputs configuration

ComponentType

Define the SIL or PIL approach

If mode of execution is simulation (`connectivity` is `sim`), choosing either value for `ComponentType` does not alter simulation results. However, `cgv.Config` recommends configuration parameter values based on the value of `ComponentType`.

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

Connectivity

Specify mode of execution

Value	Description
sim (default)	Mode of execution is simulation. Recommends changes to a subset of the configuration parameters that SIL and PIL targets require.
sil	Mode of execution is SIL. Requires that the system target file is set to 'ert.tlc' and that you do not use your own external target. Recommends changes to the configuration parameters that SIL targets require.
pil	Mode of execution is PIL with custom connectivity that you provide using the PIL Connectivity API. Recommends changes to the configuration parameters that PIL targets with custom connectivity require.

LogMode

Specify the **Signal Logging** and **Output** parameters on the **Data Import/Export** pane of the Configuration Parameters dialog box.

Value	Description
SignalLogging	Log signal data to a MATLAB workspace variable during execution. This parameter selects the Data Import/Export > Signal logging parameter in the Configuration Parameters dialog box.
SaveOutput	Save output data to a MATLAB workspace variable during execution. This parameter selects Data Import/Export > Output parameter in the Configuration Parameters dialog box.

Value	Description
	The Output parameter does not save bus outputs.

ReportOnly

The **ReportOnly** property specifies whether **cgv.Config** modifies the recommended values of the configuration parameters of the model.

If you set **ReportOnly** to **on**, **SaveModel** must be **off**.

Value	Description
off (default)	cgv.Config automatically modifies the configuration parameter values that it recommends for the model.
on	cgv.Config does not modify the configuration parameter values that it recommends for the model.

SaveModel

Specify whether to save the model with the configuration parameter values recommended by **cgv.Config**.

If you set **SaveModel** to **'on'**, **ReportOnly** must be **'off'**.

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

Methods

`configModel`

Determine and change configuration parameter values

displayReport

Display results of comparing configuration parameter values

getReportData

Return results of comparing configuration parameter values

Copy Semantics

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

Examples

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

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

See Also

`cgv.CGV`

How To

- “Programmatic Code Generation Verification”

coder.replace

Replace current MATLAB function implementation with code replacement library function in generated code

Syntax

```
coder.replace()  
coder.replace(' -errorifnoreplacement')  
coder.replace(' -warnifnoreplacement')
```

Description

`coder.replace()` replaces the current function implementation with a code replacement library function. If a match is not found in the code replacement library, code is generated without a replacement for the current function. `coder.replace` is a code generation function. It does not alter MATLAB code or MEX function generation.

During code generation, if you include `coder.replace` in a MATLAB function, `fcn`, it performs a code replacement library lookup for the following function signature:

```
[y1_type, y2_type, ..., yn_type]=fcn(x1_type, x2_type, ..., xn_type)  
y1_type, y2_type, ..., yn_type are the data types of the outputs of MATLAB  
function fcn. x1_type, x2_type, ..., xn_type are the data types of the inputs  
of fcn. coder.replace derives the output types of the function based on the  
implementation in the MATLAB function. At code generation, the contents of fcn are  
discarded and replaced with a function call that is registered in the code replacement  
library as a replacement for fcn.
```

`coder.replace(' -errorifnoreplacement')` replaces the current function implementation with a code replacement library function. If a match is not found, code generation stops. An error message describing the code replacement library lookup failure is generated.

`coder.replace(' -warnifnoreplacement')` replaces the current function implementation with a code replacement library function. If match is not found, code is generated for the current function. A warning describing the code replacement library lookup failure is generated during code generation.

Examples

Replace a MATLAB function with custom code

Replace a MATLAB function with a custom implementation that is registered in the code replacement library.

- 1 Write a MATLAB function, `calculate`, that you want to replace with a custom implementation, `replacement_calculate_impl.c`, in the generated code.

```
function y = calculate(x)
% Search in the code replacement library for replacement
% and use replacement function if available
% Error if not found
    coder.replace('-errorifnoreplacement');
    y = sqrt(x);
end
```

- 2 Write a MATLAB function, `top_function`, that calls `calculate`

```
function out = top_function(in)
    p = calculate(in);
    out = exp(p);
end
```

- 3 Create a file named `crl_table_calculate.m` that describes the function entries for a code replacement table. The replacement function `replacement_calculate_impl.c` and header file `replacement_calculate_impl.h` must be on the path.

```
hLib = RTW.TflTable;

%----- entry: calculate -----
hEnt = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(hEnt ...
    'Key', 'calculate', ...
    'Priority', 100, ...
    'ImplementationName', 'replacement_calculate_impl', ...
    'ImplementationHeaderFile', 'replacement_calculate_impl.h', ...
    'ImplementationSourceFile', 'replacement_calculate_impl.c')
% Conceptual Args

arg = getTflArgFromString(hEnt, 'y1', 'double');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(hEnt, arg);
```

```

arg = getTflArgFromString(hEnt, 'u1','double');
addConceptualArg(hEnt, arg);

% Implementation Args

arg = getTflArgFromString(hEnt, 'y1','double');
arg.IOType = 'RTW_IO_OUTPUT';
Implementation.setReturn(hEnt, arg);

arg = getTflArgFromString(hEnt, 'u1','double');
Implementation.addArgument(hEnt, arg);

%arg = getTflArgFromString(hEnt, 'y1','double*');
%arg.IOType = 'RTW_IO_OUTPUT';
%Implementation.addArgument(hEnt, arg);

addEntry(hLib, hEnt);

```

4 Create an `rtwTargetInfo` file:

```

function rtwTargetInfo(tr)
% rtwTargetInfo function to register a code
% replacement library (CRL)
% for use with codegen

% Register the CRL defined in local function locCrlRegFcn
tr.registerTargetInfo(@locCrlRegFcn);

end % End of RTWTARGETINFO

```

5 Create a `locCrlRegFcn` file:

```

function thisCrl = locCrlRegFcn

% Instantiate a CRL registry entry
thisCrl = RTW.TflRegistry;

% Define the CRL properties
thisCrl.Name = 'My calculate Example';
thisCrl.Description = 'Demonstration of function replacement';
thisCrl.TableList = {'crl_table_calculate'};
thisCrl.BaseTfl = 'C89/C90 (ANSI)';
thisCrl.TargetHWDeviceType = {'*'};

end % End of LOCCRLREGFCN

```

- 6 Refresh registration information. At the MATLAB command line, enter:

```
RTW.TargetRegistry.getInstance('reset');
```

- 7 Create a code generation configuration object.

```
cfg =coder.config('lib');
```

- 8 Specify the name of the code replacement library to use.

```
cfg.CodeReplacementLibrary='My calculate Example';
```

- 9 Generate code for `top_function` specifying that input `in` is double.

```
codegen -report -config cfg top_function -args {double(10)}
```

Because the data type of `x` and `y` is `double`, `coder.replace` searches for `double = calculate(double)` in the Code Replacement Library. If it finds a match, `codegen` generates the following code:

```
real_T top_function(real_T in)
{
    real_T p;
    p = replacement_calculate_impl(in);
    return exp(p);
}
```

In the generated code, the replacement function `replacement_calculate_impl` replaces the MATLAB function `calculate`.

- “Replace MATLAB Functions with Custom Code Using `coder.replace`”
- “Replace MATLAB Functions Specified in MATLAB Function Blocks”
- “Define Code Replacement Mappings”
- “Specify In-Place Code Replacement”
- “Register Code Replacement Mappings”

More About

Tips

- `coder.replace` is a code generation function. It does not alter MATLAB code or MEX function generation.
- Do not use multiple `coder.replace` statements inside a function.

- You cannot use `coder.replace` within conditional expressions and loops.
- `coder.replace` does not support replacements that require data alignment.
- `varargout` is not supported.
- You cannot use `coder.replace` to replace MATLAB functions that have variable-size inputs.
- `coder.replace` requires an Embedded Coder license.
- `coder.replace` disregards saturation and rounding modes when looking up function replacements in a code replacement library.
- “What Is Code Replacement?”
- “Code Replacement Libraries”
- “What Is Code Replacement Customization?”

See Also

codegen

compare

Class: `cgv.CGV`

Package: `cgv`

Compare signal data

Syntax

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

Description

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

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

`[matchNames, matchFigures, mismatchNames, mismatchFigures] = cgv.CGV.compare(data_set1, data_set2, 'Plot', 'none', 'Signals', signal_list, 'ToleranceFile', file_name)` compares only the given signals and does not produce plots.

Input Arguments

data_set1, data_set2

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

varargin

Variable number of parameter name and value pairs.

varargin Parameters

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

Plot(optional)

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

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

Signals(optional)

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

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

The syntax for multiple signal names is:

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

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

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

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

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

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

`Tolerancefile`(optional)

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

Output Arguments

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

match_names

Cell array of matching signal names.

match_figures

Array of figure handles for matching signals

mismatch_names

Cell array of mismatching signal names

mismatch_figures

Array of figure handles for mismatching signals

How To

- “Verify Numerical Equivalence with CGV”

configModel

Class: `cgv.Config`

Package: `cgv`

Determine and change configuration parameter values

Syntax

```
cfgObj.configModel()
```

Description

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

How To

- “About Model Configurations”
- “Programmatic Code Generation Verification”

checkEnvSetup

Configure your coder product to interact with Code Composer Studio

Syntax

```
checkEnvSetup(ide, boardproc, action)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3
- Texas Instruments Code Composer Studio v4
- Texas Instruments Code Composer Studio v5

Description

The `checkEnvSetup` function is only useful for validating the toolchain when the **System target file** parameter is set to `idelink_ert.tlc` or `idelink_grt.tlc`. Do not use `checkEnvSetup` when **System target** file is set to `ert.tlc`. The System target file parameter is located on the Code Generation pane in the Configuration Parameters dialog box. For more information, see “System target file”.

Before using Embedded Coder software with Texas Instruments Code Composer Studio IDE for the first time, use the `checkEnvSetup` function to verify that you have the required third-party tools, as described in:

- “Compare Version Numbers of Installed vs. Required Tools” on page 1-92
- “Set the Environment Variables” on page 1-92.

Run `checkEnvSetup` again whenever you configure CCS IDE to interact with a new board or processor, or upgrade the related third-party tools.

The syntax for this function is: `checkEnvSetup(ide, boardproc, action):`

- For the *ide* argument, enter the IDE you want to check:
 - 'ccs' checks the setup for Code Composer Studio v3
 - 'ccsv4' checks the setup for Code Composer Studio v4
 - 'ccsv5' checks the setup for Code Composer Studio v5
- For the *boardproc* argument, enter the name of a supported board or processor. You can get these names from the **Processor** parameter on the Target Hardware Resources tab (see related link at bottom of topic). For example, enter: 'F2812'.
- For the *action* argument, specify the action you want this function to perform:
 - 'list' lists the required third-party tools and version numbers.
 - 'check' lists the required third-party tools and the ones on your development system. If tools are missing, install them. If the version numbers do not match, install the required version.
 - 'setup' creates environment variables that point to the installation folders of the third-party tools. This action is required.

If your tools do not meet the requirements, the function advises you. If path information is incomplete, the function prompts you to enter path information for specific tools.

If you omit the *action* argument, the method defaults to 'setup'.

If *action* is 'list' or 'check', the checkEnvSetup function returns an output argument that contains the third-party tool information. You can assign that output argument to a variable. When *action* is 'setup', the checkEnvSetup function does not return an output argument.

Examples

Get Information About Required Tools

To find out which third-party tools your board requires, including version numbers, use 'list' as the third argument.

```
checkEnvSetup('ccs', 'F2808 eZdsp', 'list')
```

```
1. CCS (Code Composer Studio)
   Required version: 3.3.82.13
   Required for    : Automation and Code Generation
```

2. CGT (Texas Instruments C2000 Code Generation Tools)
Required version: 5.2.1
Required for : Code generation
3. DSP/BIOS (Real Time Operating System)
Required version: 5.33.05
Required for : Real-Time Data Exchange (RTDX)
4. Flash Tools (TMS320C2808 Flash APIs)
Required version: 3.02
Required for : Flash Programming
Required environment variables (name, value):
(FLASH_2808_API_INSTALLDIR, "<Flash Tools (TMS320C2808 Flash APIs) installation folder>")

Compare Version Numbers of Installed vs. Required Tools

To compare “Your version” of the installed third-party tools with the “Required version”, use 'check' as the third argument.

To resolve differences between the two version numbers, install the required software versions. Using versions of the software that are different from the required version can produce unexpected results.

```
checkEnvSetup('ccs', 'c6416', 'check')
```

1. CCS (Code Composer Studio)
Your version : 3.3.38.2
Required version: 3.3.82.13
Required for : Automation and Code Generation
2. CGT (Code Generation Tools)
Your version : 6.0.8
Required version: 6.1.10
Required for : Code generation
3. DSP/BIOS (Real Time Operating System)
Your version :
Required version: 5.33.05
Required for : Code generation
4. Texas Instruments IMGLIB (TMS320C64x)
Your version : 1.04
Required version: 1.04
Required for : CRL block replacement
C64X_IMGLIB_INSTALLDIR = "E:\apps\TexasInstruments\C6400\imglib_v104b"

Set the Environment Variables

After verifying that you have the required versions of the third-party tools, set the environment variables. Use 'setup' as the *action* argument, or omit the *action* argument.

This step is required before Embedded Coder software can use Texas Instruments Code Composer Studio to build and run an executable.


```
checkEnvSetup('ccs', 'dm6437evm')

1. Checking CCS (Code Composer Studio) version
   Required version: 3.3.82.13
   Required for    : Automation and Code Generation
   Your Version   : 3.3.38.13

2. Checking CGT (Code Generation Tools) version
   Required version: 6.1.10
   Required for    : Code generation
   Your Version   : 6.1.10

3. Checking DSP/BIOS (Real Time Operating System) version
   Required version: 5.33.05
   Required for    : Code generation
   Your Version   : 5.33.05

4. Checking Texas Instruments IMGLIB (C64x+) version
   Required version: 2.0.1
   Required for    : CRL block replacement
   Your Version   : 2.0.1
   ### Setting environment variable "C64XP_IMGLIB_INSTALLDIR"
   ### to "E:\apps\TexasInstruments\C64Plus\imglib_v201"

5. Checking DM6437EVM DVSDK (Digital Video Software Developers Kit) version
   Required version: 1.01.00.15
   Required for    : Code generation
   Your Version   : 1.01.00.15
   ### Setting environment variable "DVSDK_EVMDM6437_INSTALLDIR" to "C:\[...]"
   ### Setting environment variable "CSLR_DM6437_INSTALLDIR" to "C:\dvds[...]"
   ### Setting environment variable "PSP_EVMDM6437_INSTALLDIR" to "C:\dv[...]"
   ### Setting environment variable "NDK_INSTALL_DIR" to "C:\dvsdk_1_01_[...]"
```

More About

- “Configure the Build Process”

See Also

“System target file” | “Code Generation: Target Hardware Resources Pane”

close

Close project in IDE window

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

For the *filename* argument:

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

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

Note:

- The open method does not support the 'text' argument.

-
- Save changes to your files and projects in the IDE before you use `close`. The `close` method does not save changes, nor does it prompt you to save changes, before it closes the project.
-

Examples

To close the open project files:

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

To close the open project, `myProj`:

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

To close the active open project:

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

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

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

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

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

See Also

[add](#) | [open](#) | “[save](#)”

coder.MATLABCodeTemplate class

Package: coder

Represent code generation template for MATLAB Coder

Description

Create a `coder.MATLABCodeTemplate` object from a code generation template (CGT) file. You can use this file to customize the code generation output for MATLAB Coder™. If a CGT file is not provided, the `coder.MATLABCodeTemplate` object is created from the default template file `matlabroot/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

Construction

`newObj = coder.MATLABCodeTemplate()` creates a `coder.MATLABCodeTemplate` object from the default code generation template (CGT) file `matlabroot/toolbox/coder/matlabcoder/templates/matlabcoder_default_template.cgt`.

`newObj = coder.MATLABCodeTemplate(CGTFile)` creates a `coder.MATLABCodeTemplate` object from the code generation template file `CGTFile`. If the file is not on the MATLAB path, specify a full path to the file.

Input Arguments

CGTFile

Name of code generation template file

Methods

`emitSection`

Emit output string for template section

`getCurrentTokens`

Get current tokens

getTokenValue	Get value of token
setTokenValue	Set value of token for code generation template

Copy Semantics

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

Examples

```
newObj = coder.MATLABCodeTemplate()
newObj =
    MATLABCodeTemplate with properties:
        CGTFile: 'matlabcoder_default_template.cgt'
newObj = coder.MATLABCodeTemplate('custom_matlabcoder_template.cgt')
newObj =
    MATLABCodeTemplate with properties:
        CGTFile: 'custom_matlabcoder_template.cgt'
```

See Also

[coder.MATLABCodeTemplate.setTokenValue](#) |
[coder.MATLABCodeTemplate.getTokenValue](#) |
[coder.MATLABCodeTemplate.getCurrentTokens](#) |
[coder.MATLABCodeTemplate.emitSection](#)

Related Examples

- “Generate Custom File and Function Banners for C and C++ Code”

More About

- “Code Generation Template (CGT) Files for MATLAB”

configure

Define size and number of RTDX channel buffers

Syntax

```
configure(rx,length,num)
```

Note: `configure` produces a warning on C5000™ processors and will be removed from a future version of the software.

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`configure(rx,length,num)` sets the size of each main (host) buffer, and the number of buffers associated with *rx*. Input argument *length* is the size in bytes of each channel buffer and *num* is the number of channel buffers to create.

Main buffers must be at least 1024 bytes, with the maximum defined by the largest message. On 16-bit processors, the main buffer must be 4 bytes larger than the largest message. On 32-bit processors, set the buffer to be 8 bytes larger than the largest message. By default, `configure` creates four, 1024-byte buffers. Independent of the value of *num*, the IDE allocates one buffer for each processor.

Use CCS to check the number of buffers and the length of each one.

Examples

Create a default link to CCS and configure six main buffers of 4096 bytes each for the link.

```
IDE_Obj = ticcs           % Create the CCS link with default values.

TICCS Object:
  API version      : 1.0
  Processor type   : C67
  Processor name   : CPU
  Running?        : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

rx = rtdx(IDE_Obj)       % Create an alias to the rtdx portion.

RTDX channels      : 0

configure(rx,4096,6) % Use the alias rx to configure the length
                    % and number of buffers.
```

After you configure the buffers, use the RTDX™ tools in the IDE to verify the buffers.

See Also

readmat | readmsg | write | writemsg

connect

Connect IDE to processor

Syntax

```
IDE_Obj.connect()  
IDE_Obj.connect(debugconnection)  
IDE_Obj.connect(...,timeout)
```

IDEs

This function supports the following IDEs:

- Green Hills® MULTI®

Description

IDE_Obj.connect() connects the IDE to the processor hardware or simulator. *IDE_Obj* is the IDE handle.

IDE_Obj.connect(*debugconnection*) connects the IDE to the processor using the debug connection you specify in *debugconnection*. Enter *debugconnection* as a string enclosed in single quotation marks. *IDE_Obj* is the IDE handle. Refer to Examples to see this syntax in use.

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

Examples

The input argument `stringdebugconnection` specify the processor to connect to with the IDE. This example connects to the Freescale™ MPC5554 simulator. The `debugconnection` string is `simppc -fast -dec -rom_use_entry -cpu=ppc5554`.

```
IDE_Obj.connect('simppc -fast -dec -rom_use_entry -cpu=ppc5554')
```

See Also

`load` | `run`

copySetup

Class: `cgv.CGV`

Package: `cgv`

Create copy of `cgv.CGV` object

Syntax

```
cgvObj2 = cgvObj1.copySetup()
```

Description

`cgvObj2 = cgvObj1.copySetup()` creates a copy of a “`cgv.CGV` class” object, `cgvObj1`. The copied object, `cgvObj2`, has the same configuration as `cgvObj1`, but does not copy results of the execution.

Tips

- You can use this method to make a copy of a `cgv.CGV` object and then modify the object to run in a different mode by calling “`setMode (cgv.CGV)`”.
- If you have a `cgv.CGV` object, which reported errors or failed at execution, you can use this method to copy the object and rerun it. The copied object has the same configuration as the original object, therefore you might want to modify the location of the output files by calling “`setOutputDir (cgv.CGV)`”. Otherwise, during execution, the copied `cgv.CGV` object overwrites the output files.

Examples

Make a copy of a `cgv.CGV` object, set it to run in a different mode, then run and compare the objects in a `cgv.Batch` object.

```
cgvModel = 'rtwdemo_cgv';  
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');  
cgvObj1.run();
```

```
cgvObj2 = cgVObj1.copySetup()  
cgvObj2.setMode('sil');  
cgvObj2.run();
```

See Also

`cgv.CGV.run`

How To

- “Verify Numerical Equivalence with CGV”

copyConceptualArgsToImplementation

Copy conceptual argument specifications to matching implementation arguments for code replacement table entry

Syntax

```
copyConceptualArgsToImplementation(hEntry)
```

Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

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 code replacement table entry.

For arguments with an unsized type, such as `integer`, the code generator determines the size of the argument values based on hardware implementation configuration settings of the MATLAB code or model.

Examples

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

```
hLib = RTW.Tf1Table;
```

```
% Create an entry for addition of built-in uint8 data type
op_entry = RTW.Tf1COperationEntry;
op_entry.setTf1COperationEntryParameters( ...
    'Key', 'RTW_OP_ADD', ...
    'Priority', 90, ...
    'SaturationMode', 'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...
    'ImplementationName', 'u8_add_u8_u8', ...
    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = hLib.getTf1ArgFromString('y1','uint8');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.addConceptualArg( arg );

arg = hLib.getTf1ArgFromString('u1','uint8');
op_entry.addConceptualArg( arg );

arg = hLib.getTf1ArgFromString('u2','uint8');
op_entry.addConceptualArg( arg );

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

More About

- “Define Code Replacement Mappings”

createAndAddConceptualArg

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

Syntax

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

Input Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement table entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

argType

String specifying the argument type to create: 'RTW.Tf1ArgNumeric' for numeric or 'RTW.Tf1ArgMatrix' for matrix.

varargin

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

varargin Parameters

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

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

Name

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

IOType

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

IsSigned

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

WordLength

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

CheckSlope

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

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

DataTypeMode

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

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

DataType

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

Scaling

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

Slope

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

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

SlopeAdjustmentFactor

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

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

FixedExponent

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

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

Bias

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

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

FractionLength

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

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

BaseType

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

DimRange

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

Output Arguments

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

Description

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

Examples

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

```
op_entry = RTW.Tf1COperationEntry;
.
.
.
createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', true, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT',...
    'IsSigned', true,...
```

```

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

createAndAddConceptualArg(op_entry, 'RTW.Tf1ArgNumeric',...
    '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.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 8, ...
    'FractionLength', 0 );

% single:
createAndAddConceptualArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndAddConceptualArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndAddConceptualArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );

% Fixed-point using binary-point-only scaling:
createAndAddConceptualArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    '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.Tf1ArgNumeric', ...
    '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 “Net Slope Scaling Code Replacement” and “Equal Slope and Zero Net Bias Code Replacement” in the Embedded Coder documentation.

More About

- “Define Code Replacement Mappings”

createAndAddImplementationArg

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

Syntax

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

Input Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

argType

String specifying the argument type to create: 'RTW.Tf1ArgNumeric' for numeric.

varargin

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

varargin Parameters

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

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

Name

String specifying the argument name, for example, 'u1'.

IOType

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

IsSigned

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

WordLength

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

DataTypeMode

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

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

DataType

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

Scaling

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

Slope

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

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

SlopeAdjustmentFactor

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

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

FixedExponent

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

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

Bias

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

FractionLength

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

Value

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

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

You can inject constant input arguments into the implementation signature for code replacement table entries, but if the argument values or the number of arguments required depends on compile-time information, you should use custom matching. For more information, see “Customize Matching and Replacement Process for Functions”.

Output Arguments

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

Description

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

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

Examples

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.Tf1COperationEntry;
.
.
.
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'y1', ...
    'IOType', 'RTW_IO_OUTPUT', ...
    'IsSigned', true, ...
    'WordLength', 32, ...
    'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT',...
    'IsSigned', true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    '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.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
    'IsSigned', false, ...
    'WordLength', 8, ...
    'FractionLength', 0 );

% single:
createAndAddImplementationArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name', 'u1', ...
    'IOType', 'RTW_IO_INPUT', ...
```



```
        'DataTypeMode', 'single' );

% double:
createAndAddImplementationArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndAddImplementationArg(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name',          'u1', ...
    'IOType',        'RTW_IO_INPUT', ...
    'DataTypeMode', 'boolean' );
```

More About

- “Define Code Replacement Mappings”

See Also

createAndSetCImplementationReturn

createAndSetCImplementationReturn

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

Syntax

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

Input Arguments

hEntry

Handle to a code replacement table entry previously returned by instantiating a code replacement entry class, such as *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = RTW.Tf1COperationEntry.

argType

String specifying the argument type to create: 'RTW.Tf1ArgNumeric' 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.

Output Arguments

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

Description

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

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).

Examples

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.Tf1COperationEntry;  
.  
.  
.  
createAndSetCImplementationReturn(op_entry, 'RTW.Tf1ArgNumeric', ...
```

```

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

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    'Name',      'u1', ...
    'IOType',   'RTW_IO_INPUT',...
    'IsSigned', true,...
    'WordLength', 32, ...
    'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.Tf1ArgNumeric',...
    '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.Tf1ArgNumeric', ...
    'Name',      'y1', ...
    'IOType',   'RTW_IO_OUTPUT', ...
    'IsSigned', false, ...
    'WordLength', 8, ...
    'FractionLength', 0 );

% single:
createAndSetCImplementationReturn(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'y1', ...
    'IOType',   'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'single' );

% double:
createAndSetCImplementationReturn(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'y1', ...
    'IOType',   'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'double' );

% boolean:
createAndSetCImplementationReturn(hEntry, 'RTW.Tf1ArgNumeric', ...
    'Name',      'y1', ...
    'IOType',   'RTW_IO_OUTPUT', ...
    'DataTypeMode', 'boolean' );

```

More About

- “Define Code Replacement Mappings”

See Also

createAndAddImplementationArg

createToleranceFile

Class: `cgv.CGV`

Package: `cgv`

Create file correlating tolerance information with signal names

Syntax

```
cgvObj.createToleranceFile(file_name , signal_list, tolerance_list)
```

Description

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

Input Arguments

file_name

Name for the file containing the tolerance specification for each signal. Use this file as input to `cgv.CGV.compare` and `cgv.Batch.addTest`.

signal_list

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

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

The syntax for multiple signal names is:

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

```
'log_data.block_name.Data(:,3)',...  
'log_data.block_name.Data(:,4)');
```

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

```
signal_list = {'global_tolerance',...  
'log_data.block_name.Data(:,1)',...  
'log_data.block_name.Data(:,2)'};  
  
tolerance_list = {'relative', 0.02},...  
{'relative', 0.015},{'absolute', 0.05}};
```

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

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

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

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


---


```

tolerance_list

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

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

How To

- “Verify Numerical Equivalence with CGV”

dir

Files and folders in current IDE window

Syntax

```
dir(IDE_Obj)
d = dir(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

- `d(3)` returns the third element in the structure.
- `d(10)` returns the tenth element in the structure `d`.
- `d(4).date` returns the `date` field value for the fourth structure element.

See Also

`"cd"` | `open`

disable

Disable RTDX interface, specified channel, or RTDX channels

Note: Support for `disable` on C5000 processors will be removed in a future version.

Syntax

```
disable(rx, 'channel')
disable(rx, 'all')
disable(rx)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`disable(rx, 'channel')` disables the open channel specified by the string `channel`, for `rx`. Input argument `rx` represents the RTDX portion of the associated link to the IDE.

`disable(rx, 'all')` disables the open channels associated with `rx`.

`disable(rx)` disables the RTDX interface for `rx`.

Important Requirements for Using `disable`

On the processor side, `disable` depends on RTDX to disable channels or the interface. To use `disable`, meet the following requirements:

- 1 The processor must be running a program.

- 2 You enabled the RTDX interface.
- 3 Your processor program polls periodically.

Examples

When you have opened and used channels to communicate with a processor, disable the channels and RTDX before ending your session. Use `disable` to switch off open channels and disable RTDX, as follows:

```
disable(rtdx(IDE_Obj),'all') % Disable the open RTDX channels.  
disable(rtdx(IDE_Obj))      % Disable RTDX interface.
```

See Also

`close` | `enable` | `open`

display (IDE Object)

Properties of IDE handle

Syntax

```
display(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

```
display(IDE_Obj)
```

IDE Object:

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

See Also

`get`

display

Generate message that describes how to open code execution profiling report

Syntax

```
myExecutionProfile  
myExecutionProfile.display
```

Description

myExecutionProfile or *myExecutionProfile*.display generates a message that describes how you can open the code execution profiling report.

myExecutionProfile is a workspace variable, specified through the configuration parameter `CodeExecutionProfileVariable` and generated by a simulation.

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”

See Also

report

displayReport

Class: `cgv.Config`

Package: `cgv`

Display results of comparing configuration parameter values

Syntax

```
cfgObj.displayReport()
```

Description

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

How To

- “Verify Numerical Equivalence Between Two Modes of Execution of a Model”

coder.MATLABCodeTemplate.emitSection

Class: coder.MATLABCodeTemplate

Package: coder

Emit output string for template section

Syntax

```
emitSection(sectionName,isCPPComment)
```

Description

`emitSection(sectionName,isCPPComment)` emits the output string for the code template section that `sectionName` specifies. If `isCPPComment` is `true`, `emitSection` uses C++ style comments. If `emitSection` is `false`, it uses C style comments. Use `emitSection` to preview banners before you generate code. Before invoking `emitSection` to emit the banner for a template section, you must set the values for all tokens used in that section.

Input Arguments

sectionName

Name of template section specified as a string.

Specify `sectionName` as one of the following:

'FileBanner'	'VariableDeclarationsBanner'
'FunctionBanner'	'VariableDefinitionsBanner'
'SharedUtilityBanner'	'FunctionDeclarationsBanner'
'FileTrailer'	'FunctionDefinitionsBanner'
'IncludeFilesBanner'	'CustomSourceCodeBanner'
'TypeDefinitionsBanner'	'CustomHeaderCodeBanner'


```
'NamedConstantsBanner'
```

isCplusplusComment

C++ style comments flag specified as a Boolean.

Specify `true` for C++ style comments. Specify `false` for C style comments.

Examples

Emit File Banner from Default Template

This example shows how to set the `FileName` token value and emit the default file banner.

Create a `coder.MATLABCodeTemplate` object from the default template.

```
newObj = coder.MATLABCodeTemplate
```

Set the `FileName` token value.

```
fileN = 'myfilename.c';
newObj.setTokenValue('FileName', fileN)
```

Emit the file banner.

```
newObj.emitSection('FileBanner', false)
```

The `emitSection` method generates the file banner replacing the `FileName` token with the file name that you specified. It replaces the `MATLABCoderVersion` token with the current MATLAB Coder version number. It replaces the `SourceGeneratedOn` token with the time stamp.

```
/*
 * File: myfilename.c
 *
 * MATLAB Coder version           : 2.7
 * C/C++ source code generated on : 07-Apr-2014 17:43:32
 */
```

Emit Include Files Banner from Custom Template

This example shows how to create and modify a custom code generation template (CGT) file. It shows how to emit the include files section banner from the custom CGT file.

Create a local copy of the default CGT file for MATLAB Coder. Name it `myCGTFile.cgt`.

In your local copy of the CGT File, in the `IncludeFilesBanner` open tag, change the style to "box".

```
<IncludeFilesBanner style="box">
Include Files
</IncludeFilesBanner>
```

Create a `MATLABCodeTemplate` object from your custom CGT file.

```
CGTFile = 'myCGTFile.cgt';
newObj = coder.MATLABCodeTemplate(CGTFile);
```

Emit the include files section banner using C++ style comments.

```
newObj.emitSection('IncludeFilesBanner', true)
```

The `emitSection` method generates the include files section banner using the box style with C++ style comments.

```
////////////////////////////////////
// Include Files
////////////////////////////////////
```

- “Generate Custom File and Function Banners for C and C++ Code”

See Also

`coder.MATLABCodeTemplate.setTokenValue` |
`coder.MATLABCodeTemplate.getTokenValue` |
`coder.MATLABCodeTemplate.getCurrentTokens`

More About

- “Code Generation Template (CGT) Files for MATLAB”

enable

Enable RTDX interface, specified channel, or RTDX channels

Note: Support for `enable` on C5000 processors will be removed in a future version.

Syntax

```
enable(rx, 'channel')  
enable(rx, 'all')  
enable(rx)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`enable(rx, 'channel')` enables the open channel specified by the string `channel`, for RTDX link `rx`. The input argument `rx` represents the RTDX portion of the associated link to the IDE.

`enable(rx, 'all')` enables the open channels associated with `rx`.

`enable(rx)` enables the RTDX interface for `rx`.

Important Requirements for Using enable

On the processor side, `enable` depends on RTDX to enable channels. To use `enable`, meet the following requirements:

- 1 The processor must be running a program when you enable the RTDX interface. When the processor is not running, the state defaults to disabled.
- 2 Enable the RTDX interface before you enable individual channels.
- 3 Channels must be open.
- 4 Your processor program must poll periodically.
- 5 Using code in the program running on the processor to enable channels overrides the default disabled state of the channels.

Examples

To use channels to RTDX, you must both open and enable the channels:

```
IDE_Obj = ticcs; % Create a new connection to the IDE.  
enable(rtdx(IDE_Obj)) % Enable the RTDX interface.  
open(rtdx(IDE_Obj), 'inputchannel', 'w') % Open a channel for sending  
                                     % data to the processor.  
enable(rtdx(IDE_Obj), 'inputchannel') % Enable the channel so you can use  
                                     % it.
```

See Also

disable | open

enableCPP

Enable C++ support for function entry in code replacement table

Syntax

```
enableCPP(hEntry)
```

Arguments

hEntry

Handle to a code replacement function entry previously returned by *hEntry* = `RTW.Tf1CFunctionEntry` or *hEntry* = `MyCustomFunctionEntry`, where `MyCustomFunctionEntry` is a class derived from `RTW.Tf1CFunctionEntry`.

Description

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

Note: When you register a code replacement library containing C++ function entries, you must specify the value { 'C++' } for the `LanguageConstraint` property of the code replacement registry entry. For more information, see “Register Code Replacement Mappings”.

Examples

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

```
fcn_entry = RTW.Tf1CFunctionEntry;  
fcn_entry.setTf1CFunctionEntryParameters( ...
```

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

More About

- “Math Function Code Replacement”
- “Define Code Replacement Mappings”

See Also

registerCPPFunctionEntry | setNameSpace

excludeCheck

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Exclude checks

Syntax

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

Description

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

Input Arguments

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

Examples

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

```
excludeCheck(obj, 'mathworks.codegen.CodeInstrumentation');
```

See Also

Simulink.ModelAdvisor

How To

- “Create Custom Objectives”
- “About IDs”

flush

Flush data or messages from specified RTDX channels

Note: flush support for C5000 processors will be removed in a future version.

Syntax

```
flush(rx,channel,num,timeout)
flush(rx,channel,num)
flush(rx,channel,[],timeout)
flush(rx,channel)
flush(rx,'all')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`flush(rx,channel,num,timeout)` removes *num* oldest data messages from the RTDX channel queue specified by *channel* in *rx*. To determine how long to wait for the function to complete, `flush` uses *timeout* (in seconds) rather than the global timeout period stored in *rx*. `flush` applies the timeout processing when it flushes the last message in the channel queue, because the flush function performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx,channel,num)` removes the *num* oldest messages from the RTDX channel queue in *rx* specified by the string *channel*. `flush` uses the global timeout period stored in *rx* to determine how long to wait for the process to complete. Compare this to the

previous syntax that specifies the timeout period. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx, channel, [], timeout)` removes the data messages from the RTDX channel queue specified by *channel* in *rx*. To determine how long to wait for the function to complete, `flush` uses *timeout* (in seconds) rather than the global timeout period stored in *rx*. `flush` applies the timeout processing when it flushes the last message in the channel queue, because `flush` performs a read to advance the read pointer past the last message. Use this calling syntax only when you specify a channel configured for read access.

`flush(rx, channel)` removes the pending data messages from the RTDX channel queue specified by *channel* in *rx*. Unlike the preceding syntax options, you use this statement to remove messages for both read-configured and write-configured channels.

`flush(rx, 'all')` removes the data messages from the RTDX channel queues.

When you use `flush` with a write-configured RTDX channel, your coder product sends the messages in the write queue to the processor. For read-configured channels, `flush` removes one or more messages from the queue depending on the input argument *num* you supply and disposes of them.

Examples

To show how to use `flush`, this example writes data to the processor over the input channel, then uses `flush` to remove a message from the read queue for the output channel:

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx, 'ichan', 'w');
enable(rx, 'ichan');
open(rx, 'ochan', 'r');
enable(rx, 'ochan');
indata = 1:10;
writemsg(rx, 'ichan', int16(indata));
flush(rx, 'ochan', 1);
```

Now flush the remaining messages from the read channel:

```
flush(rx, 'ochan', 'all');
```

See Also

enable | open

getArgCategory

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.getClassInterfaceSpecification(modelName)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

<i>category</i>	String specifying the argument category — 'Value', 'Pointer', or 'Reference' — for the specified Simulink model port.
-----------------	---

Alternatives

To view argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to display step method argument categories. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getArgCategory

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>category</i>	String specifying the argument category, 'Value' or 'Pointer', for the specified Simulink model port.
-----------------	---

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument categories. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

getArgName

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <i>obj</i> = <code>RTW.getClassInterfaceSpecification(<i>modelName</i>)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.

Output Arguments

<i>argName</i>	String specifying the argument name for the specified Simulink model port.
----------------	--

Alternatives

To view argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to display step method argument names. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getArgName

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>argName</i>	String specifying the argument name for the specified Simulink model port.
----------------	--

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument names. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

getArgPosition

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>position</i>	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0.
-----------------	---

Alternatives

To view argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to display step method argument positions. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getArgPosition

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>position</i>	Integer specifying the argument position — 1 for first, 2 for second, etc. — for the specified Simulink model port. Without an argument for the specified port, the function returns 0.
-----------------	---

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument positions. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

getArgQualifier

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>qualifier</i>	String specifying the argument type qualifier — 'none', 'const', 'const *', 'const * const', or 'const &' — for the specified Simulink model port.
------------------	--

Alternatives

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

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getArgQualifier

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>qualifier</i>	String specifying the argument type qualifier — 'none', 'const', 'const *', or 'const * const' — for the specified Simulink model port.
------------------	---

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get argument qualifiers. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

getbuildopt

Generate structure of build tools and options

Syntax

```
bt = getbuildopt(IDE_Obj)  
cs = getbuildopt(IDE_Obj, file)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

`bt = getbuildopt(IDE_Obj)` returns an array of structures in `bt`. Each structure includes an entry for each defined build tool. This list of build tools comes from the active project and active build configuration. Included in the structure is a string that describes the command-line tool options. `bt` uses the following format for elements in the structures:

- `bt(n).name` — Name of the build tool.
- `bt(n).optstring` — command-line switches for build tool in `bt(n)`.

`cs = getbuildopt(IDE_Obj, file)` returns a string of build options for the source file specified by *file*. *file* must exist in the active project. The resulting `cs` string comes from the active build configuration. The type of source file (from the file extension) defines the build tool used by the `cs` string.

getClassName

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <i>obj</i> = RTW.getClassInterfaceSpecification (<i>modelName</i>).
------------	--

Output Arguments

<i>clsName</i>	A string specifying the name of the class described by the specified model-specific C++ class interface.
----------------	--

Alternatives

To view the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which

displays the model class name and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getCoderExecutionProfile

Extract execution time profile for MATLAB function

Syntax

```
myExecutionProfile=getCoderExecutionProfile('myMATLABFunction');
```

Description

myExecutionProfile=getCoderExecutionProfile('myMATLABFunction'); creates a workspace variable that contains the execution time profile of your MATLAB function.

Run the command after the completion and termination of the SIL/PIL execution of your MATLAB function.

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

Sections | TimerTicksPerSecond | report

coder.MATLABCodeTemplate.getCurrentTokens

Class: coder.MATLABCodeTemplate

Package: coder

Get current tokens

Syntax

```
currentTokens = getCurrentTokens()
```

Description

`currentTokens = getCurrentTokens()` returns list of current tokens in the `MATLABCodeTemplate` object

Output Arguments

currentTokens — Current tokens

cell array of strings

A list of current tokens in the `MATLABCodeTemplate` object, returned as a cell array of strings.

Examples

Create a `MATLABCodeTemplate` object with the default template, then list its tokens.

```
newObj = coder.MATLABCodeTemplate;  
% Creates a MATLABCodeTemplate object from the default template  
newObj.getCurrentTokens()  
% Returns a list of tokens for the template
```

See Also

`coder.MATLABCodeTemplate.setTokenValue` |
`coder.MATLABCodeTemplate.getTokenValue` |
`coder.MATLABCodeTemplate.emitSection`

Related Examples

- “Generate Custom File and Function Banners for C and C++ Code”

More About

- “Code Generation Template (CGT) Files for MATLAB”

getDefaultConf

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

`getDefaultConf(obj)`

Description

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

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

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPArgsClass</code> or <code>obj = RTW.ModelCPPVoidClass</code> .
------------	---

Alternatives

To view C++ class interface default configuration information in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click

the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the **I/O arguments step method** view of this dialog box, click the **Get Default Configuration** button to display default configuration information. In the **void-void step method** view, you can see the default configuration information without clicking a button. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getDefaultConf

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

`getDefaultConf(obj)`

Description

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

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

Input Arguments

<i>obj</i>	Handle to a model-specific C prototype function control object previously returned by <i>obj</i> = <code>RTW.ModelSpecificCPrototype</code> .
------------	---

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get the default configuration. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

getFunctionName

Class: RTW.ModelSpecificCPrototype

Package: RTW

Get function name from model-specific C function prototype

Syntax

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

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get function names. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

Name

Get name of profiled code section

Syntax

```
SectionName = NthSectionProfile.Name
```

Description

SectionName = *NthSectionProfile*.Name returns the name that identifies the profiled code section.

The software generates an identifier based on the model entity that corresponds to the profiled section of code.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

SectionName

Name that identifies profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum`

| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Name

Get name of profiled code section

Syntax

```
SectionName = NthSectionProfile.Name
```

Description

SectionName = *NthSectionProfile*.Name returns the name that identifies the profiled code section.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

SectionName

Name that identifies profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

```
getCoderExecutionProfile | TimerTicksPerSecond | NumCalls  
| MaximumSelfTimeCallNum | TotalExecutionTimeInTicks  
| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |  
TotalTurnaroundTimeInTicks | Sections | report | Number  
| MaximumExecutionTimeCallNum | ExecutionTimeInTicks |  
MaximumExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks |  
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum
```

getNamespace

Class: RTW.ModelCPPClass

Package: RTW

Get namespace from model-specific C++ class interface

Syntax

```
nsName = getNamespace(obj)
```

Description

nsName = getNamespace(*obj*) gets the namespace of the class described by the specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <i>obj</i> = RTW.getClassInterfaceSpecification (<i>modelName</i>).
------------	--

Output Arguments

<i>nsName</i>	A string specifying the namespace of the class described by the specified model-specific C++ class interface.
---------------	---

Alternatives

To view the model namespace in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which

displays the model class name and namespace and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getNumArgs

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

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

Description

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

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.getClassInterfaceSpecification(modelName)</code> .
------------	---

Output Arguments

num	An integer specifying the number of step method arguments.
-----	--

Alternatives

To view the number of step method arguments in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class.

In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to display the step method arguments. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getNumArgs

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

num = getNumArgs(*obj*)

Description

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

Input Arguments

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

Output Arguments

num An integer specifying the number of function arguments.

Alternatives

Click the **Get Default Configuration** button in the Model Interface dialog box to get arguments. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

NumCalls

Total number of calls to profiled code section

Syntax

```
TotalNumCalls = NthSectionProfile.NumCalls
```

Description

TotalNumCalls = *NthSectionProfile*.NumCalls returns the total number of calls to the profiled code section over the entire simulation.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

TotalNumCalls

Total number of calls

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name |
Number | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum
| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |

MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

NumCalls

Total number of calls to profiled code section

Syntax

```
TotalNumCalls = NthSectionProfile.NumCalls
```

Description

TotalNumCalls = *NthSectionProfile*.NumCalls returns the total number of calls to the profiled code section over the entire execution.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalNumCalls

Total number of calls

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks` | `MaximumSelfTimeInTicks` | `TurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks` | `Sections` | `report` | `Name` | `Number` | `MaximumExecutionTimeCallNum` | `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` | `SelfTimeInTicks` | `TotalSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum`

getOutputData

Class: `cgv.CGV`

Package: `cgv`

Get output data

Syntax

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

Description

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

How To

- “Verify Numerical Equivalence with CGV”

getPreview

Class: RTW.ModelSpecificCPrototype

Package: RTW

Get model-specific C function prototype code preview

Syntax

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

Description

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

Input Arguments

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

Output Arguments

<i>preview</i>	String specifying the function prototype for the step or initialization function.
----------------	---

Alternatives

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

How To

- “Function Prototype Control”

getReportData

Class: `cgv.Config`

Package: `cgv`

Return results of comparing configuration parameter values

Syntax

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

Description

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

How To

- “Verify Numerical Equivalence with CGV”

getSavedSignals

Class: `cgv.CGV`

Package: `cgv`

Display list of signal names to command line

Syntax

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

Description

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

Tips

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

How To

- “Verify Numerical Equivalence with CGV”

Number

Get number that uniquely identifies profiled code section

Syntax

```
SectionNumber = NthSectionProfile.Number
```

Description

SectionNumber = *NthSectionProfile*.Number returns a number that uniquely identifies the profiled code section, for example, in the code execution profiling report.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

SectionNumber

Number of profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

```
Sections | TimerTicksPerSecond | display | report | Name |  
NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum  
| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |  
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks  
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
```

MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

Number

Get number that uniquely identifies profiled code section

Syntax

```
SectionNumber = NthSectionProfile.Number
```

Description

SectionNumber = *NthSectionProfile*.Number returns a number that uniquely identifies the profiled code section.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

SectionNumber

Number of profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

```
getCoderExecutionProfile | TimerTicksPerSecond | NumCalls  
| MaximumSelfTimeCallNum | TotalExecutionTimeInTicks  
| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |  
TotalTurnaroundTimeInTicks | Sections | report | Name |  
MaximumExecutionTimeCallNum | ExecutionTimeInTicks |  
MaximumExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks |  
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum
```

Sections

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)  
numberOfSections = length(myExecutionProfile.Sections)
```

Description

NthSectionProfile = *myExecutionProfile*.Sections(*N*) returns an `coder.profile.ExecutionTimeSection` object for the *N*th profiled code section.

numberOfSections = length(*myExecutionProfile*.Sections) returns the number of code sections for which profile data is available.

myExecutionProfile is a workspace variable generated by a simulation.

Input Arguments

N

Index of code section for which profile data is required

Output Arguments

NthSectionProfile

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.
- `Number` — Number of the code section.
- `NumCalls` — Number of calls to the code section.

- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire simulation.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire simulation.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire simulation.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

numberOfSections

Number of code sections with profile data

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`TimerTicksPerSecond` | `display` | `report`

Sections

Get array of `coder.profile.ExecutionTimeSection` objects for profiled code sections

Syntax

```
NthSectionProfile = myExecutionProfile.Sections(N)  
numberOfSections = length(myExecutionProfile.Sections)
```

Description

NthSectionProfile = *myExecutionProfile*.Sections(*N*) returns an `coder.profile.ExecutionTimeSection` object for the *N*th profiled code section.

numberOfSections = length(*myExecutionProfile*.Sections) returns the number of code sections for which profile data is available.

myExecutionProfile is a workspace variable that you create using `getCoderExecutionProfile`.

Input Arguments

N

Index of code section for which profile data is required

Output Arguments

NthSectionProfile

Object that contains profile information about the code section. You can use the following `coder.profile.ExecutionTimeSection` methods to retrieve the information:

- `Name` — Name of the code section.
- `Number` — Number of the code section.

- `NumCalls` — Number of calls to the code section.
- `TotalExecutionTimeInTicks` — Total number of timer ticks recorded for the code section over the entire execution.
- `TurnaroundTimeInTicks` — Time between start and finish of the code section, in timer ticks.
- `TotalTurnaroundTimeInTicks` — Total number of timer ticks between start and finish of the code section, over the entire execution.
- `MaximumExecutionTimeInTicks` — Maximum number of timer ticks for a single invocation of the code section.
- `MaximumExecutionTimeCallNum` — Number of call associated with the maximum number of timer ticks recorded for a single invocation of the code section.
- `MaximumTurnaroundTimeInTicks` — Maximum number of ticks between start and finish for a single invocation.
- `MaximumTurnaroundTimeCallNum` — Number of call associated with the maximum time between start and finish of a single invocation.
- `MaximumSelfTimeInTicks` — Maximum self time, in timer ticks.
- `SelfTimeInTicks` — Self time for the code section, in timer ticks.
- `TotalSelfTimeInTicks` — Total self time for the code section, over the entire execution.
- `MaximumSelfTimeCallNum` — Call associated with maximum self time.
- `ExecutionTimeInTicks` — Vector of execution times.

numberOfSections

Number of code sections with profile data

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `report`

getStatus

Class: `cgv.CGV`

Package: `cgv`

Return execution status

Syntax

```
status = cgvObj.getStatus()  
status = cgvObj.getStatus(inputName)
```

Description

`status = cgvObj.getStatus()` returns the execution status of *cgvObj*. *cgvObj* is a handle to a `cgv.CGV` object.

`status = cgvObj.getStatus(inputName)` returns the status of a single execution for *inputName*.

Input Arguments

inputName

inputName is a unique numeric or character identifier associated with input data, which is added to the `cgv.CGV` object using “`addInputData (cgv.CGV)`”.

Output Arguments

status

If *inputName* is provided, *status* is the result of the execution of input data associated with *inputName*.

Value	Description
none	Execution has not run.

Value	Description
pending	Execution is currently running.
completed	Execution ran to completion without errors and output data is available.
passed	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned no differences.
error	Execution produced an error.
failed	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned a difference.

If inputName is not provided, the following pseudocode describes the return status:

```

if (all executions return 'passed')
  status = 'passed'
else if (all executions return 'passed' or 'completed')
  status = 'completed'
else if (an execution returns 'error')
  status = 'error'
else if (an execution returns 'failed')
  status = 'failed'
else if (an execution returns 'none' or 'pending')
  status = 'none'

```

See Also

cgv.CGV.addInputData | cgv.CGV.run | cgv.CGV.addBaseline

How To

- “Verify Numerical Equivalence with CGV”

getStepMethodName

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

```
fcnName = getStepMethodName(obj)
```

Description

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

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.getClassInterfaceSpecification(<i>modelName</i>)</code> .
------------	--

Output Arguments

<i>fcnName</i>	A string specifying the name of the step method described by the specified model-specific C++ class interface.
----------------	--

Alternatives

To view the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, which

displays the step method name and allows you to display and configure the step method for your model class. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

getTflArgFromString

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

Syntax

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

Input Arguments

hTable

Handle to a code replacement table previously returned by *hTable* = RTW.TflTable.

name

String specifying the name to use for the code replacement argument, for example, 'y1'.

datatype

String specifying a built-in data type or a fixed-point data type to use for the code replacement argument:

- Valid built-in data types are 'integer', 'int8', 'int16', 'int32', 'long', 'long_long', 'uinteger', 'uint8', 'uint16', 'uint32', 'ulong', 'ulong_long', 'single', 'double', 'boolean', and 'logical'.
- You can specify fixed-point data types using the `fixdt` function from Fixed-Point Designer™ software; for example, 'fixdt(1,16,2)'.

Output Arguments

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

Description

The `getTflArgFromString` function creates a code replacement 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.

Examples

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 code replacement table entry.

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

More About

- “Define Code Replacement Mappings”

See Also

`addConceptualArg`

getTf1DWorkFromString

Create code replacement DWork argument for semaphore entry based on specified name and data type

Syntax

```
arg = getTf1DWorkFromString(hTable, name, datatype)
```

Input Arguments

hTable

Handle to a code replacement table previously returned by *hTable* = RTW.Tf1Table.

name

String specifying the name to use for the code replacement DWork argument, for example, 'd1'.

datatype

String specifying a data type to use for the code replacement DWork argument. Currently, you must specify 'void* '.

Output Arguments

Handle to the created code replacement argument, which can be specified to the addDWorkArg function. See the example below.

Description

The getTf1DWorkFromString function creates a code replacement DWork argument, based on a specified name and data type, for a semaphore entry in a code replacement table.

Examples

In the following example, `getTf1DWorkFromString` is used to create a `void*` argument named `d1`. The argument is then added as a `DWork` argument for a semaphore entry in a code replacement table.

```
hLib = RTW.Tf1Table;
sem_entry = RTW.Tf1CSemaphoreEntry;
.
.
.
% DWork Arg

arg = hLib.getTf1DWorkFromString('d1','void*');
sem_entry.addDWorkArg( arg );
.
.
.
hLib.addEntry( sem_entry );
```

More About

- “Semaphore and Mutex Function Replacement”
- “Define Code Replacement Mappings”

See Also

`addDWorkArg`

RTW.Tf1BlasEntryGenerator

Create code replacement table entry for a BLAS operation

Syntax

```
obj = RTW.Tf1BlasEntryGenerator
```

Description

obj = RTW.Tf1BlasEntryGenerator creates a handle, *obj*, to a code replacement table entry for a BLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Output Arguments

obj Handle to code replacement table entry for a BLAS operator.

Examples

Create a code replacement table entry for a BLAS operator, hEnt.

```
hEnt = RTW.Tf1BlasEntryGenerator;
```

See Also

RTW.Tf1CBlasEntryGenerator | RTW.Tf1COperationEntry | RTW.Tf1Table

How To

- “Define Code Replacement Mappings”
- “Matrix Multiplication Operation to MathWorks BLAS Code Replacement”

RTW.Tf1CBlasEntryGenerator

Create code replacement table entry for a CBLAS operation

Syntax

```
obj = RTW.Tf1CBlasEntryGenerator
```

Description

obj = RTW.Tf1CBlasEntryGenerator creates a handle, *obj*, to a code replacement table entry for a CBLAS operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Output Arguments

obj Handle to code replacement table entry for a CBLAS operator.

Examples

Create a code replacement table entry for a CBLAS operator, hEnt.

```
hEnt = RTW.Tf1CBlasEntryGenerator;
```

See Also

RTW.Tf1BlasEntryGenerator | RTW.Tf1COperationEntry | RTW.Tf1Table

How To

- “Define Code Replacement Mappings”
- “Matrix Multiplication Operation to ANSI/ISO C BLAS Code Replacement”

RTW.Tf1CFunctionEntry

Create code replacement table entry for a function

Syntax

```
obj = RTW.Tf1CFunctionEntry
```

Description

obj = RTW.Tf1CFunctionEntry creates a handle, *obj*, to a code replacement table entry for a function. The entry maps a conceptual representation of a function to an implementation (replacement) representation.

Output Arguments

obj Handle to code replacement table entry for a function.

Examples

Create a code replacement table entry for a function, hEnt.

```
hEnt = RTW.Tf1CFunctionEntry;
```

See Also

RTW.Tf1CFunctionEntryML | | RTW.Tf1Table

How To

- “Define Code Replacement Mappings”
- “Math Function Code Replacement”
- “Memory Function Code Replacement”
- “Nonfinite Function Code Replacement”

- “Lookup Table Function Code Replacement”
-

RTW.Tf1CFunctionEntryML

Base class for custom code replacement table function entry

Syntax

RTW.Tf1CFunctionEntryML

Description

Derive a class from RTW.Tf1CFunctionEntryML to represent your custom function entry.

Examples

“Customize Matching and Replacement Process for Functions”

See Also

RTW.Tf1CFunctionEntry | RTW.Tf1Table

How To

- “Define Code Replacement Mappings”
- “Customize Matching and Replacement Process for Functions”

RTW.Tf1COperationEntry

Create code replacement table entry for an operator

Syntax

```
obj = RTW.Tf1COperationEntry
```

Description

obj = RTW.Tf1COperationEntry creates a handle, *obj*, to a code replacement table entry for an operator. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Output Arguments

obj Handle to code replacement table entry for an operator.

Examples

Create a code replacement table entry for an operator, hEnt.

```
hEnt = RTW.Tf1COperationEntry;
```

See Also

| | RTW.Tf1Table | RTW.Tf1COperationEntryGenerator |
RTW.Tf1COperationEntryGenerator_NetSlope | RTW.Tf1COperationEntryML

How To

- “Define Code Replacement Mappings”
- “Scalar Operator Code Replacement”
- “Addition and Subtraction Operator Code Replacement”

- “Small Matrix Operation to Processor Code Replacement”

RTW.Tf1COperationEntryGenerator

Create code replacement table entry for a fixed-point addition or subtraction operation

Syntax

```
obj = RTW.Tf1COperationEntryGenerator
```

Description

obj = RTW.Tf1COperationEntryGenerator creates a handle, *obj*, to a code replacement table entry for a fixed-point addition or subtraction operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Output Arguments

<i>obj</i>	Handle to code replacement table entry for a fixed-point addition or subtraction operation.
------------	---

Examples

Create a code replacement table entry for a fixed-point addition or subtraction operation, *hEnt*.

```
hEnt = RTW.Tf1COperationEntryGenerator;
```

See Also

RTW.Tf1COperationEntry | RTW.Tf1Table |
RTW.Tf1COperationEntryGenerator_NetSlope | RTW.Tf1COperationEntryML

How To

- “Define Code Replacement Mappings”

- “Fixed-Point Operator Code Replacement”
- “Binary-Point-Only Scaling Code Replacement”
- “Slope Bias Scaling Code Replacement”

RTW.Tf1COperationEntryGenerator_NetSlope

Create code replacement table entry for a net slope fixed-point operation

Syntax

```
obj = RTW.Tf1COperationEntryGenerator_NetSlope
```

Description

obj = RTW.Tf1COperationEntryGenerator_NetSlope creates a handle, *obj*, to a code replacement table entry for a net slope fixed-point operation. The entry maps a conceptual representation of an operator to an implementation (replacement) representation.

Output Arguments

<i>obj</i>	Handle to code replacement table entry for a net slope fixed-point operation.
------------	---

Examples

Create a code replacement table entry for a net slope fixed-point operation, *hEnt*.

```
hEnt = RTW.Tf1COperationEntryGenerator_NetSlope;
```

See Also

RTW.Tf1COperationEntry | RTW.Tf1COperationEntryGenerator | RTW.Tf1COperationEntryML

How To

- “Define Code Replacement Mappings”
- “Fixed-Point Operator Code Replacement”

- “Net Slope Scaling Code Replacement”
- “Equal Slope and Zero Net Bias Code Replacement”

RTW.Tf1COperationEntryML

Base class for custom code replacement table operator entry

Syntax

RTW.Tf1COperationEntryML

Description

Derive a class from RTW.Tf1COperationEntryML to represent your custom operator entry.

Examples

“Customize Matching and Replacement Process for Operators”

See Also

RTW.Tf1COperationEntry | RTW.Tf1Table

How To

- “Define Code Replacement Mappings”
- “Customize Matching and Replacement Process for Operators”

RTW.Tf1CSemaphoreEntry

Create code replacement table entry for a semaphore or mutex

Syntax

```
obj = RTW.Tf1CSemaphoreEntry
```

Description

obj = RTW.Tf1CSemaphoreEntry creates a handle, *obj*, to a code replacement table entry for a semaphore or mutex. The entry maps a conceptual representation of a semaphore or mutex to an implementation (replacement) representation.

Output Arguments

obj Handle to code replacement table entry for a semaphore or mutex.

Examples

Create a code replacement table entry for a semaphore or mutex, hEnt.

```
hEnt = RTW.Tf1CSemaphoreEntry;
```

How To

- “Define Code Replacement Mappings”
- “Semaphore and Mutex Function Replacement”

RTW.Tf1Table

Create code replacement table

Syntax

```
obj = RTW.Tf1Table
```

Description

obj = RTW.Tf1Table creates a handle, *obj*, to a code replacement table.

Output Arguments

obj Handle to code replacement table.

Examples

Create a code replacement table object, *hTable*.

```
hTable = RTW.Tf1Table;
```

How To

- “Define Code Replacement Mappings”

Time

Get simulation time for code section

Syntax

```
SimTime = NthSectionProfile.Time
```

Description

`SimTime = NthSectionProfile.Time` returns a simulation time vector that corresponds to the execution time measurements for the code section.

Examples

Get Simulation Time for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;  
set_param('rtwdemo_sil_topmodel',...  
          'CodeExecutionProfiling', 'on');  
set_param('rtwdemo_sil_topmodel',...  
          'SimulationMode', 'software-in-the-loop (SIL)');  
set_param('rtwdemo_sil_topmodel',...  
          'CodeProfilingInstrumentation', 'on');  
set_param('rtwdemo_sil_topmodel',...  
          'CodeProfilingSaveOptions', 'AllData');  
sim('rtwdemo_sil_topmodel');  
The simulation generates the workspace variable executionProfile (default).
```

At the end of the simulation, get profile for the seventh code section.

```
seventhSectionProfile = executionProfile.Sections(7);
```

Get vector representing simulation time for code section.

```
simulationTimeVector = seventhSectionProfile.Time;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection`

object

Object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

SimTime — Simulation time

double

Simulation time, in seconds, for section of code. Returned as a vector.

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “Analyze Code Execution Data”

See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` | `Sections`

Time

Time over which code section execution time measurements are made

Syntax

```
Time = NthSectionProfile.Time
```

Description

`Time = NthSectionProfile.Time` returns a time vector corresponding to the period over which execution times are measured for the code section.

Examples

Get Time Vector for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot, 'toolbox', 'coder', 'examples', 'kalman');

copyfile(fullfile(src_dir, 'kalman01.m'), '.')
copyfile(fullfile(src_dir, 'test01_ui.m'), '.')
copyfile(fullfile(src_dir, 'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir, 'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;

codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');

coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```


At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
   Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
secondSectionProfile = executionProfile.Sections(2);
```

Get time vector for code section.

```
time = secondSectionProfile.Time;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection`

object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

Time — `Time`

double

Time, in seconds, over which measurements are made for code section. Returned as a vector.

More About

- “Generate Execution Time Profile”

- “Analyze Execution Time Data”

See Also

`ExecutionTimeInSeconds` | `ExecutionTimeInTicks` |
`getCoderExecutionProfile` | `Sections`

timeline

Display invocations of code sections over execution timeline

Syntax

```
timeline(executionProfile)
```

Description

`timeline(executionProfile)` displays invocations of each profiled code section over the execution timeline.

Examples

Display Code Section Invocations

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;  
set_param('rtwdemo_sil_topmodel',...  
          'CodeExecutionProfiling', 'on');  
set_param('rtwdemo_sil_topmodel',...  
          'SimulationMode', 'software-in-the-loop (SIL)');  
set_param('rtwdemo_sil_topmodel',...  
          'CodeProfilingInstrumentation', 'on');  
set_param('rtwdemo_sil_topmodel',...  
          'CodeProfilingSaveOptions', 'AllData');  
sim('rtwdemo_sil_topmodel');
```

























The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, open a code execution report.

```
report(executionProfile)
```

Under **Profiled Sections of Code**, in the **Model** column, expand all nodes. You see profile information for eight code sections. For example, the task `rtwdemo_sil_topmodel_step` and functions `CounterTypeA` and `CounterTypeB`.

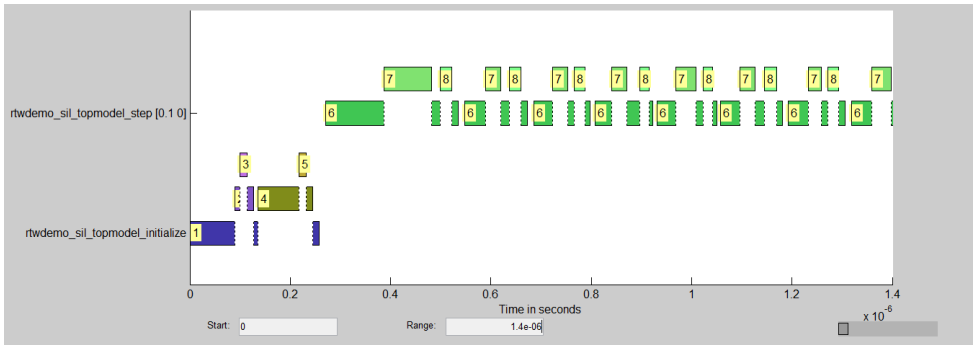
2. Profiled Sections of Code

Model	Maximum Execution Time	Average Execution Time	Maximum Self Time	Average Self Time	Calls	
[-] rtwdemo_sil_topmodel_initialize	257	257	111	111	1	  
[-] CounterTypeA	38	38	23	23	1	  
CounterTypeA	15	15	15	15	1	  
[-] CounterTypeB	109	109	94	94	1	  
CounterTypeB	15	15	15	15	1	  
[-] rtwdemo_sil_topmodel_step [0.1 0]	265	121	147	61	101	  
CounterTypeA	94	36	94	36	101	  
CounterTypeB	37	24	37	24	101	  


Display code section invocations.

```
timeline(executionProfile)
```

In the Execution Profile window, you see numbered horizontal bars that represent invocations of the code sections.



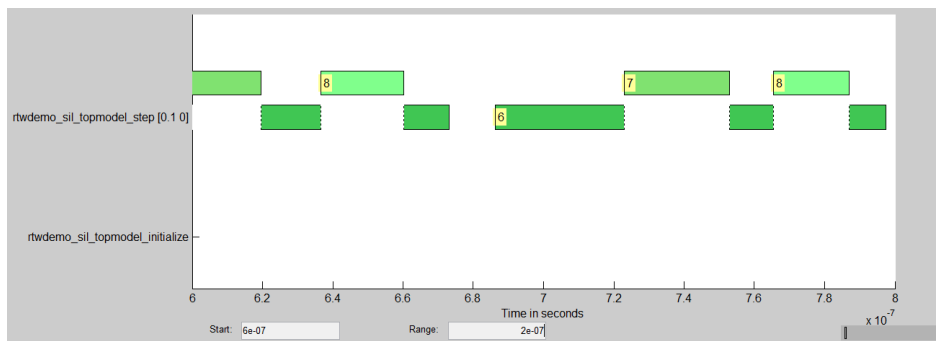
For example, the blue bars show when the first section, `rtwdemo_sil_topmodel_initialize`, is invoked.

To see the first code section, in the first row of the Code Execution Profiling Report, click the icon .

The Code Generation Report displays the function call.

```
64 PROFILE_START_TASK_SECTION(10);
65 rtwdemo_sil_topmodel_initialize();
66 PROFILE_END_TASK_SECTION(10);
```

To see what code sections are invoked over a specific time period, use the **Start** and **Range** fields of the Execution Profile window. For example, in the **Start** and **Range** fields, enter **6e-07** and **2e-07** respectively. Then press **Enter**.



Between 0.6 μ s and 0.8 μ s, you see that the task `rtwdemo_sil_topmodel_step` (code section 6) and the functions `CounterTypeA` (code section 7) and `CounterTypeB` (code section 8) are invoked.

On the bottom right of the Execution Profile window, the indicator shows what portion of the execution timeline is being displayed.

Input Arguments

executionProfile — `coder.profile.ExecutionTime` object

When you run a simulation with code execution profiling, the software generates `executionProfile` as a workspace variable.

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”

See Also

“report”

TimerTicksPerSecond

Get and set number of timer ticks per second

Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond  
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

Description

timerTicksPerSecVal = *myExecutionProfile*.TimerTicksPerSecond returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is 10^6 .

myExecutionProfile.TimerTicksPerSecond = *timerTicksPerSecVal* sets the number of timer ticks per second. Use this method if the “Create PIL Target Connectivity Configuration” does not specify this value.

myExecutionProfile is a workspace variable generated by a simulation.

Tip You can calculate the execution time in seconds using the formula

ExecutionTimeInSecs = *ExecutionTimeInTicks* / *TimerTicksPerSecond* .

Input Arguments

timerTicksPerSecVal

Number of timer ticks per second

Output Arguments

timerTicksPerSecVal

Number of timer ticks per second

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | report | Name | Number | display | NumCalls |
MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum
| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

TimerTicksPerSecond

Get and set number of timer ticks per second

Syntax

```
timerTicksPerSecVal = myExecutionProfile.TimerTicksPerSecond  
myExecutionProfile.TimerTicksPerSecond = timerTicksPerSecVal
```

Description

timerTicksPerSecVal = *myExecutionProfile*.TimerTicksPerSecond returns the number of timer ticks per second. For example, if the timer runs at 1 MHz, then the number of ticks per second is 10^6 .

myExecutionProfile.TimerTicksPerSecond = *timerTicksPerSecVal* sets the number of timer ticks per second. Use this method if the target connectivity configuration does not specify this value.

myExecutionProfile is a workspace variable that you create using `getCoderExecutionProfile`.

Tip You can calculate the execution time in seconds using the formula

$ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond$.

Input Arguments

timerTicksPerSecVal

Number of timer ticks per second

Output Arguments

timerTicksPerSecVal

Number of timer ticks per second

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”
- “Create PIL Target Connectivity Configuration”

See Also

`getCoderExecutionProfile` | `NumCalls` | `MaximumSelfTimeCallNum`
| `TotalExecutionTimeInTicks` | `MaximumSelfTimeInTicks` |
`TurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks` | `Sections` | `report`
| `Name` | `Number` | `MaximumExecutionTimeCallNum` | `ExecutionTimeInTicks` |
`MaximumExecutionTimeInTicks` | `SelfTimeInTicks` | `TotalSelfTimeInTicks` |
`MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum`

coder.MATLABCodeTemplate.getTokenValue

Class: coder.MATLABCodeTemplate

Package: coder

Get value of token

Syntax

```
tokenValue = getTokenValue(tokenName)
```

Description

`tokenValue = getTokenValue(tokenName)` returns the value of the specified token.

Input Arguments

tokenName

Name of token

Default: empty

Output Arguments

tokenValue — Token value

string

The current value of `tokenName`, returned as a string.

Examples

Create a `MATLABCodeTemplate` object with the default template, then get the value for a token.

```
newObj = coder.MATLABCodeTemplate;  
% Creates a MATLABCodeTemplate object from the default template  
newObj.getCurrentTokens()  
% Get list of current tokens  
newObj.getTokenValue('MATLABCoderVersion')  
% Check value of a token
```

See Also

coder.MATLABCodeTemplate.setTokenValue |
coder.MATLABCodeTemplate.getCurrentTokens |
coder.MATLABCodeTemplate.emitSection

Related Examples

- “Generate Custom File and Function Banners for C and C++ Code”

More About

- “Code Generation Template (CGT) Files for MATLAB”

halt

Halt program execution by processor

Syntax

```
halt(IDE_Obj)  
halt(IDE_Obj, timeout)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

Examples

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

```
isrunning(IDE_Obj)

ans =

     1

halt(IDE_Obj) % Stop the running application on the processor.
isrunning(IDE_Obj)

ans =

     0
```

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

See Also

`isrunning` | “reset” | `run`

info

Information about processor

Syntax

```
adf = info(IDE_Obj)
adf = info(IDE_Obj)
adf = info(rx)
adf = info(IDE_Obj)
adf = info(rx)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

`adf = info(IDE_Obj)` returns debugger or processor properties associated with the IDE handle object, `IDE_Obj`.

Using info with Multiprocessor Boards

For multiprocessor targets, the `info` method returns properties for each processor with the array.

Examples

Using `info` with `IDE_Obj`, which is associated with 1 processor:

```
oinfo = info(IDE_Obj);
```

Using info with `IDE_Obj`, which is associated with 2 processors:

```
oinfo = info(IDE_Obj); % Returns a 1x2 array of info struct
```

Using info with CCS IDE

`adf = info(IDE_Obj)` returns the property names and property values associated with the processor accessed by `IDE_Obj`. `adf` is a structure containing the following information elements and values.

Structure Element	Data Type	Description
<code>adf.procname</code>	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with <code>IDE_Obj</code> .
<code>adf.isbigendian</code>	Boolean	Value describing the byte ordering used by the processor. When the processor is big-endian, this value is 1. Little-endian processors return 0.
<code>adf.family</code>	Integer	Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors.
<code>adf.subfamily</code>	Decimal	Decimal representation of the hexadecimal identification value that TI assigns to the processor to identify the processor subfamily. IDs range from 0x000 to 0x3822. Use <code>dec2hex</code> to convert the value in <code>adf.subfamily</code> to standard notation. For example <code>dec2hex(adf.subfamily)</code> produces '67' when the processor is a member of the 67xx processor family.
<code>adf.timeout</code>	Integer	Default timeout value MATLAB software uses when transferring data to and from CCS. Functions that use a timeout value have an optional <i>timeout</i> input argument. When you omit the optional argument, MATLAB software uses 10s as the default value.

`adf = info(rx)` returns `info` as a cell array containing the names of your open RTDX channels.

Examples

On a PC with a simulator configured in CCS IDE, `info` returns the configuration for the processor being simulated:

```
info(IDE_Obj)

ans =

    procname: 'CPU'
  isbigendian: 0
         family: 320
    subfamily: 103
         timeout: 10
```

This example simulates the TMS320C62xx processor running in little-endian mode. When you use CCS Setup Utility to change the processor from little-endian to big-endian, `info` shows the change.

```
info(IDE_Obj)

ans =

    procname: 'CPU'
  isbigendian: 1
         family: 320
    subfamily: 103
         timeout: 10
```

If you have two open channels, `chan1` and `chan2`,

```
adf = info(rx)

returns

adf =
'chan1'
'chan2'
```

where `adf` is a cell array. You can dereference the entries in `adf` to manipulate the channels. For example, you can close a channel by dereferencing the channel in `adf` in the `close` function syntax.

```
close(rx.adf{1,1})
```


Using info with VisualDSP++ IDE

`adf = info(IDE_Obj)` returns the property names and property values associated with the processor accessed by `IDE_Obj`. The `adf` variable is a structure containing the following information elements and values.

Structure Element	Data Type	Description
<code>adf.procname</code>	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with <code>IDE_Obj</code> .
<code>adf.proctype</code>	String	String with the type of the DSP processor. The type property is the processor type like "ADSP-21065L" or "ADSP-2181".
<code>adf.revision</code>	String	String with the silicon revision string of the processor.

`adf = info(rx)` returns `info` as a cell array containing the names of your open RTDX channels.

Examples

When you have an `adivdsp` object `IDE_Obj`, `info` provides information about the object:

```
IDE_Obj = adivdsp('sessionname', 'Testsession')
```

ADIVDSP Object:

```
Session name      : Testsession
Processor name    : ADSP-BF533
Processor type    : ADSP-BF533
Processor number  : 0
Default timeout   : 10.00 secs
```

```
objinfo = info(IDE_Obj)
```

```
objinfo =
```

```
    procname: 'ADSP-BF533'
    proctype: 'ADSP-BF533'
    revision: ''
```

```
objinfo.procname
```

ans =

ADSP-BF533

See Also

dec2hex | get | set

insert

Insert debug point in file

Syntax

```
insert(IDE_Obj,addr,type,timeout)
insert(IDE_Obj,addr)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

See Also

address | run

isenabled

Determine whether RTDX link is enabled for communications

Note: Support for `isenabled` on C5000 processors will be removed in a future version.

Syntax

```
isenabled(rx, 'channel')  
isenabled(rx)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`isenabled(rx, 'channel')` returns `ans = 1` when the RTDX channel specified by string `'channel'` is enabled for read or write communications. When `'channel'` has not been enabled, `isenabled` returns `ans = 0`.

`isenabled(rx)` returns `ans = 1` when RTDX has been enabled, independent of a channel. When you have not enabled RTDX you get `ans = 0` back.

Important Requirements for Using `isenabled`

On the processor side, `isenabled` depends on RTDX to determine and report the RTDX status. Therefore the you must meet the following requirements to use `isenabled`.

- 1 The processor must be running a program when you query the RTDX interface.

- 2 You must enable the RTDX interface before you check the status of individual channels or the interface.
- 3 Your processor program must be polling periodically for `isenabled` to work.

Note: For `isenabled` to return valid results, your processor must be running a loaded program. When the processor is not running, `isenabled` returns a status that may not represent the true state of the channels or RTDX.

Examples

With a program loaded on your processor, you can determine whether RTDX channels are ready for use. Restart your program to be sure it is running. The processor must be running for `isenabled` and `enabled` to function. This example creates a `ticcs` object `IDE_Obj` to begin.

```
restart(IDE_Obj)
run(IDE_Obj, 'run');
rtdx.enable(IDE_Obj, 'ichan');
rtdx.isenabled(IDE_Obj, 'ichan')
```

MATLAB software returns 1 indicating that your channel 'ichan' is enabled for RTDX communications. To determine the mode for the channel, use `rtdx(IDE_Obj)` to display the object properties.

See Also

`clear` | `disable` | `enable`

isreadable

Determine whether specified memory block can read MATLAB software

Note: Support for `isreadable(rx, 'channel')` on C5000 processors will be removed in a future version.

Syntax

```
isreadable(IDE_Obj, address, 'datatype', count)
isreadable(IDE_Obj, address, 'datatype')
isreadable(rx, 'channel')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`isreadable(IDE_Obj, address, 'datatype', count)` returns 1 if the processor referred to by `IDE_Obj` can read the memory block defined by the `address`, `count`, and `datatype` input arguments. When the processor cannot read a portion of the specified memory block, `isreadable` returns 0. You use the same memory block specification for this function as you use for the `read` function.

The data block being tested begins at the memory location defined by `address`. `count` determines the number of values to be read. `datatype` defines the format of data stored in the memory block. `isreadable` uses the `datatype` string to determine the number of bytes to read per stored value. For details about each input parameter, read the following descriptions.

`address` — `isreadable` uses `address` to define the beginning of the memory block to read. You provide values for `address` as either decimal or hexadecimal representations

of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector [*location*, *page*], a string, or a decimal value.

When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ticcs` sets the page to 0 at creation if you omit the page property as an input argument. For processors that have one memory page, setting the page value to 0 lets you specify memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
'1F'	String	Location is 31 decimal on the page referred to by <code>page(IDE_Obj)</code>
10	Decimal	Address is 10 decimal on the page referred to by <code>page(IDE_Obj)</code>
[18,1]	Vector	Address location 10 decimal on memory page 1 (<code>page(IDE_Obj) = 1</code>)

To specify the address in hexadecimal format, enter the *address* property value as a string. `isreadable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. When you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `page(IDE_Obj)`.

count — A numeric scalar or vector that defines the number of *datatype* values to test for being readable. To produce parallel structure with `read`, *count* can be a vector to define multidimensional data blocks. This function tests a block of data whose size is the product of the dimensions of the input vector.

datatype — A string that represents a MATLAB software data type. The total memory block size is derived from the value of *count* and the *datatype* you specify. *datatype* determines how many bytes to check for each memory value. `isreadable` supports the following data types.

datatype String	Number of Bytes/Value	Description
'double'	8	Double-precision floating point values

datatype String	Number of Bytes/ Value	Description
'int8'	1	Signed 8-bit integers
'int16'	2	Signed 16-bit integers
'int32'	4	Signed 32-bit integers
'single'	4	Single-precision floating point data
'uint8'	1	Unsigned 8-bit integers
'uint16'	2	Unsigned 16-bit integers
'uint32'	4	Unsigned 32-bit integers

Like the `iswritable`, `write`, and `read` functions, `isreadable` checks for valid address values. Illegal address values would be an address space larger than the available space for the processor:

- 2^{32} for the C6xxx series
- 2^{16} for the C5xxx series

When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`isreadable(IDE_Obj, address, 'datatype')` returns 1 if the processor referred to by `IDE_Obj` can read the memory block defined by the `address`, and `datatype` input arguments. When the processor cannot read a portion of the specified memory block, `isreadable` returns 0. Notice that you use the same memory block specification for this function as you use for the `read` function. The data block being tested begins at the memory location defined by `address`. When you omit the `count` option, `count` defaults to one.

`isreadable(rx, 'channel')` returns a 1 when the RTDX channel specified by the string `channel`, associated with link `rx`, is configured for `read` operation. When `channel` is not configured for reading, `isreadable` returns 0.

Like the `iswritable`, `read`, and `write` functions, `isreadable` checks for valid address values. Illegal address values are address spaces larger than the available space for the processor:

- 2^{32} for the C6xxx series

- 2^{16} for the C5xxx series

When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

Note: `isreadable` relies on the memory map option in the IDE. If you did not define the memory map for the processor in the IDE, `isreadable` does not produce useful results. Refer to your Texas Instruments Code Composer Studio documentation for information on configuring memory maps.

Examples

When you write scripts to run models in the MATLAB environment and the IDE, the `isreadable` function is very useful. Use `isreadable` to check that the channel from which you are reading is configured.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);

% Define read and write channels to the processor linked by IDE_Obj.
open(rx, 'ichannel', 'r');
open(rx, 'ochannel', 'w');
enable(rx, 'ochannel');
enable(rx, 'ichannel');

isreadable(rx, 'ochannel')
ans =
    0
isreadable(rx, 'ichannel')
ans =
    1
```

Now that your script knows that it can read from `ichannel`, it proceeds to read messages as required.

See Also

`hex2dec` | `iswritable` | `read`

isrtdxcapable

Determine whether processor supports RTDX

Note: Support for `isrtdxcapable` on C5000 processors will be removed in a future version.

Syntax

```
b = isrtdxcapable(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`b = isrtdxcapable(IDE_Obj)` returns `b = 1` when the processor referenced by object *IDE_Obj* supports RTDX. When the processor does not support RTDX, `isrtdxcapable` returns `b = 0`.

Using isrtdxcapable with Multiprocessor Boards

When your board contains more than one processor, `isrtdxcapable` checks each processor on the processor, as defined by the *IDE_Obj* object, and returns the RTDX capability for each processor on the board. In the returned variable `b`, you find a vector that contains the information for each accessed processor.

Examples

Create a link to your C6711 DSK. Test to see if the processor on the board supports RTDX.

```
IDE_Obj = ticcs; %Assumes you have one board and it is the C6711 DSK.  
b = isrtdxcapable(IDE_Obj)  
b =  
    1
```

isrunning

Determine whether processor is executing process

Syntax

```
isrunning(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

Examples

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

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

```
ans =
```

```
1  
halt(IDE_Obj)  
isrunning(IDE_Obj)
```

ans =

0

See Also

halt | load | run

isvisible

Determine whether IDE appears on desktop

Syntax

```
isvisible(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

`isvisible(IDE_Obj)` returns 1 if the IDE is running on the desktop and the window is open. If the IDE is not running or is running in the background, this method returns 0.

Examples

First use a constructor to create an IDE handle object and start the IDE. To determine if the IDE is visible:

```
isvisible(IDE_Obj) #determine if the ide is visible  
  
ans =  
  
    1  
visible(IDE_Obj,0) #make the ide invisible  
isvisible(IDE_Obj) #determine if the ide is visible  
  
ans =  
  
    0
```

Notice that the IDE is not visible on your desktop. Recall that MATLAB software did not open the IDE. When you close MATLAB software with the IDE in this invisible state, the IDE remains running in the background. To close it, perform either of the following tasks:

- Open MATLAB software. Create a link to the IDE. Use the new link to make the IDE visible. Close the IDE.
- Open Microsoft® Windows® Task Manager. Click **Processes**. Find and highlight `IDE_Obj_app.exe`. Click **End Task**.

See Also

`info` | `visible`

iswritable

Determine whether MATLAB can write to specified memory block

Note: Support for `iswritable(rx, 'channel')` on C5000 processors will be removed in a future version.

Syntax

```
iswritable(IDE_Obj, address, 'datatype', count)
iswritable(IDE_Obj, address, 'datatype')
iswritable(rx, 'channel')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`iswritable(IDE_Obj, address, 'datatype', count)` returns 1 if MATLAB software can write to the memory block defined by the `address`, `count`, and `datatype` input arguments on the processor referred to by `IDE_Obj`. When the processor cannot write to a portion of the specified memory block, `iswritable` returns 0. You use the same memory block specification for this function as you use for the `write` function.

The data block being tested begins at the memory location defined by `address`. `count` determines the number of values to write. `datatype` defines the format of data stored in the memory block. `iswritable` uses the `datatype` parameter to determine the number of bytes to write per stored value. For details about each input parameter, read the following descriptions.

address — `iswritable` uses **address** to define the beginning of the memory block to write to. You provide values for **address** as either decimal or hexadecimal representations of a memory location in the processor. The full address at a memory location consists of two parts: the offset and the memory page, entered as a vector [*Location*, *page*], a string, or a decimal value. When the processor has only one memory page, as is true for many digital signal processors, the page portion of the memory address is 0. By default, `ticcs` sets the page to 0 at creation if you omit the page property as an input argument.

For processors that have one memory page, setting the page value to 0 lets you specify memory locations in the processor using the memory location without the page value.

Examples of Address Property Values

Property Value	Address Type	Interpretation
1F	String	Location is 31 decimal on the page referred to by <code>page (IDE_ Obj)</code>
10	Decimal	Address is 10 decimal on the page referred to by <code>page (IDE_ Obj)</code>
[18,1]	Vector	Address location 10 decimal on memory page 1 (<code>page (IDE_ Obj) = 1</code>)

To specify the address in hexadecimal format, enter the address property value as a string. `iswritable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. When you use the string option to enter the address as a hex value, you cannot specify the memory page. For string input, the memory page defaults to the page specified by `page (IDE_ Obj)`.

count — A numeric scalar or vector that defines the number of **datatype** values to test for being writable. To produce parallel structure with `write`, **count** can be a vector to define multidimensional data blocks. This function tests a block of data whose size is the total number of elements in matrix specified by the input vector. If **count** is the vector [10 10 10], then:

```
iswritable(IDE_Obj,31,[10 10 10])
```

`iswritable` writes 1000 values (10*10*10) to the processor. For a two-dimensional matrix defined with **count** as

```
iswritable(IDE_Obj,31,[5 6])
```

`iswritable` writes 30 values to the processor.

`datatype` — a string that represents a MATLAB data type. The total memory block size is derived from the value of `count` and the specified `datatype`. `datatype` determines how many bytes to check for each memory value. `iswritable` supports the following data types.

datatype String	Description
'double'	Double-precision floating point values
'int8'	Signed 8-bit integers
'int16'	Signed 16-bit integers
'int32'	Signed 32-bit integers
'single'	Single-precision floating point data
'uint8'	Unsigned 8-bit integers
'uint16'	Unsigned 16-bit integers
'uint32'	Unsigned 32-bit integers

`iswritable(IDE_Obj, address, 'datatype')` returns 1 if the processor referred to by `IDE_Obj` can write to the memory block defined by the `address`, and `count` input arguments. When the processor cannot write a portion of the specified memory block, `iswritable` returns 0. Notice that you use the same memory block specification for this function as you use for the `write` function. The data block tested begins at the memory location defined by `address`. When you omit the `count` option, `count` defaults to one.

Note: `iswritable` relies on the memory map option in the IDE. If you did not define the memory map for the processor in the IDE, this function does not produce useful results. Refer to your Texas Instruments Code Composer Studio documentation for information on configuring memory maps.

Like the `isreadable`, `read`, and `write` functions, `iswritable` checks for valid address values. Illegal address values would be an address space larger than the available space for the processor:

- 2^{32} for the C6xxx series

- 2^{16} for the C5xxx series

When the function identifies an illegal address, it returns an error message stating that the address values are out of range.

`iswritable(rx, 'channel')` returns a Boolean value signifying whether the RTDX channel specified by `channel` and `rx`, is configured for write operations.

Examples

When you write scripts to run models in MATLAB software and the IDE, the `iswritable` function is very useful. Use `iswritable` to check that the channel to which you are writing to is indeed configured.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);

% Define read and write channels to the processor linked by IDE_Obj.
open(rx, 'ichannel', 'r');
open(rx, 'ochannel', 'w');
enable(rx, 'ochannel');
enable(rx, 'ichannel');

iswritable(rx, 'ochannel')
ans =
     1
iswritable(rx, 'ichannel')
ans =
     0
```

Now that your script knows that it can write to `'ichannel'`, it proceeds to write messages as required.

See Also

`hex2dec` | `isreadable` | `read`

list

Information listings from IDE

Syntax

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

Using list with CCS IDE

`infolist = list(IDE_Obj, type)` reads information about your CCS session and returns it in `infolist`. Different types of information and return formats apply depending on the input arguments you supply to the `list` function call. The `type` argument specifies which information listing to return. To determine the information that `list` returns, use one of the following as the `type` parameter string:

- **project** — Tell `list` to return information about the current project in CCS.
- **variable** — Tell `list` to return information about one or more embedded variables.
- **globalvar** — Tell `list` to return information about one or more global embedded variables.
- **function** — Tell `list` to return details about one or more functions in your project.

The `list` function returns dynamic CCS information that can be altered by the user. Returned listings represent snapshots of the current CCS configuration only. Be aware that earlier copies of `infolist` might contain stale information.

Also, `list` may report incorrect information when you make changes to variables from MATLAB software. To report variable information, `list` uses the CCS API, which only knows about variables in CCS. Your changes from MATLAB software do not appear

through the API and `list`. For example, the following operations return incorrect or old data information from `list`.

Suppose your original prototype is

```
unsigned short tgtFunction7(signed short signedShortArray1[]);
```

After creating the function object `fcnObj`, perform a `declare` operation with this string to change the declaration:

```
unsigned short tgtFunction7(unsigned short signedShortArray1[]);
```

Now try using `list` to return information about `signedShortArray1`.

```
list(fcnObj, 'signedShortArray1')

  address: [3442 1]
location: [1x66 char]
   size: 1
  bitsize: 16
  reftype: 'short'
  referent: [1x1 struct]
member_pts_to_same_struct: 0
   name: 'signedShortArray1'
```

You get this outcome because `list` uses the CCS API to query information about a particular variable. As far as the API is concerned, the first input variable is a `short*`. Changing the declaration does not change anything.

When you specify option `type` as **project**, for example `infolist = list('project')`, the method returns a vector of structures that contain project information in the following format.

infolist Structure Element	Description
<code>infolist(1).name</code>	Project file name (with path).
<code>infolist(1).type</code>	Project type — <code>project</code> , <code>projlib</code> , or <code>projectx</code> , refer to <code>new</code> .
<code>infolist(1).proccesortype</code>	String description of processor CPU.
<code>infolist(1).srcfiles</code>	Vector of structures that describes project source files. Each structure contains the name and path for each source file — <code>infolist(1).srcfiles.name</code> .

infolist Structure Element	Description
<code>infolist(1).buildcfg</code>	Vector of structures that describe build configurations, each with the following entries: <ul style="list-style-type: none"> • <code>infolist(1).buildcfg.name</code> — the build configuration name. • <code>infolist(1).buildcfg.outpath</code> — the default folder for storing the build output.
<code>infolist(2)....</code>	...
<code>infolist(n)....</code>	...

`infolist = list(IDE_Obj, 'variable')` returns a structure of structures that contains information on the local variables within scope. The list also includes information on the global variables. However, that if a local variable has the same symbol name as a global variable, list returns the information about the local variable.

`infolist = list(IDE_Obj, 'variable', varname)` returns information about the specified variable `varname`.

`infolist = list(IDE_Obj, 'variable', varnamelist)` returns information about variables in a list specified by `varnamelist`. The information returned in each structure follows the following format when you specify option `type` as **variable**.

infolist Structure Element	Description
<code>infolist.varname(1).name</code>	Symbol name.
<code>infolist.varname(1).isglobal</code>	Indicates whether symbol is global or local.
<code>infolist.varname(1).location</code>	Information about the location of the symbol.
<code>infolist.varname(1).size</code>	Size per dimension.
<code>infolist.varname(1).uclass</code>	<code>ticcs</code> object class that matches the type of this symbol.
<code>infolist.varname(1).bitsize</code>	Size in bits. More information is added to the structure depending on the symbol type.
<code>infolist.varname(2)....</code>	...
<code>infolist.varname(n)....</code>	...

`list` uses the variable name as the field name to refer to the structure information for the variable.

`infolist = list(IDE_Obj, 'globalvar')` returns a structure that contains information on the global variables.

`infolist = list(IDE_Obj, 'globalvar',varname)` returns a structure that contains information on the specified global variable.

`infolist = list(IDE_Obj, 'globalvar',varnamelist)` returns a structure that contains information on global variables in the list. The returned information follows the same format as the syntax `infolist = list(IDE_Obj, 'variable',...)`.

`infolist = list(IDE_Obj, 'function')` returns a structure that contains information on the functions in the embedded program.

`infolist = list(IDE_Obj, 'function',functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = list(IDE_Obj, 'function',functionnamelist)` returns a structure that contains information on the specified functions in `functionnamelist`. The returned information follows the following format when you specify option `type` as **function**.

infolist Structure Element	Description
<code>infolist.functionname(1).name</code>	Function name
<code>infolist.functionname(1).filename</code>	Name of file where function is defined
<code>infolist.functionname(1).address</code>	Relevant address information such as start address and end address
<code>infolist.functionname(1).funcvar</code>	Variables local to the function
<code>infolist.functionname(1).uclass</code>	<code>ticcs</code> object class that matches the type of this symbol — function
<code>infolist.functionname(1).funcdecl</code>	Function declaration — where information such as the function return type is contained

infolist Structure Element	Description
<code>infolist.functionname(1).islibfunc</code>	Determine if the library is a function
<code>infolist.functionname(1).linepos</code>	Start and end line positions of function
<code>infolist.functionname(1).funcinfo</code>	Miscellaneous information about the function
<code>infolist.functionname(2)...</code>	...
<code>infolist.functionname(n)...</code>	...

To refer to the function structure information, `list` uses the function name as the field name.

The following list provides important information about variable and field names:

- When a variable name, type name, or function name is not a valid MATLAB software structure field name, `list` replaces or modifies the name so it becomes valid.
- In field names that contain the invalid dollar character \$, `list` replaces the \$ with DOLLAR.
- Changing the MATLAB software field name does not change the name of the embedded symbol or type.

Examples

To show how to use `list` with a defined C type, variable `typename1` includes the `type` argument. Because valid field names cannot contain the \$ character, `list` changes the \$ to DOLLAR.

```
typename1 = '$fake3'; % name of defined C type with no tag
list(IDE_Obj, 'type', typename1);
ans =
```

```
    DOLLARfake0 : [typeinfo]
```

```
ans.DOLLARfake0 =
```

```
    type: 'struct $fake0'
    size: 1
    uclass: 'structure'
```

```
    sizeof: 1  
    members: [1x1 struct]
```

When you request information about a project in CCS, you see a listing like the following that includes structures containing details about your project.

```
projectinfo = list(IDE_Obj, 'project')  
  
projectinfo =  
    name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'  
    type: 'project'  
    processor: 'TMS320C67XX'  
    srcfiles: [69x1 struct]  
    buildcfg: [3x1 struct]
```

See Also
info

listsessions

List existing sessions

Syntax

```
list = listsessions  
list = listsessions('verbose')
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++

Description

`list = listsessions` returns `list` that contains a listing of the sessions by name currently in the development environment.

`list = listsessions('verbose')` adds the optional input argument `verbose`. When you include the `verbose` argument, `listsessions` returns a cell array that contains one row for each existing session. Each row has three columns — processor type, platform name, and processor name.

See Also

`adivdsp`

load

Load program file onto processor

Syntax

```
load(IDE_Obj, filename, timeout)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

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

Examples

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

See Also

“cd” | dir | open

ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

If you set the `CodeProfilingSaveOptions` parameter to 'SummaryOnly', `NthSectionProfile.ExecutionTimeInSeconds` returns an empty array. To change that parameter, open the Configuration Parameters dialog box by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to All measurement and analysis data.

Examples

Get Execution Times for Code Section

Run a simulation with a model that is configured to generate a workspace variable with execution time measurements.

```
rtwdemo_sil_topmodel;  
set_param('rtwdemo_sil_topmodel', 'CodeExecutionProfiling', 'on');  
set_param('rtwdemo_sil_topmodel', 'SimulationMode', 'software-in-the-loop (SIL)');  
set_param('rtwdemo_sil_topmodel', 'CodeProfilingInstrumentation', 'on');  
set_param('rtwdemo_sil_topmodel', 'CodeProfilingSaveOptions', 'AllData');  
sim('rtwdemo_sil_topmodel');
```

The simulation generates the workspace variable `executionProfile` (default).

At the end of the simulation, get the profile for the seventh code section.

```
SeventhSectionProfile = executionProfile.Sections(7);
```

Get vector of execution times for the code section.

```
time_vector = SeventhSectionProfile.ExecutionTimeInSeconds;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection`

object

Object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

ExecutionTimes — Execution time measurements

double

Execution times, in seconds, for section of code. Returned as a vector.

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “Analyze Code Execution Data”

See Also

`ExecutionTimeInTicks` | `Sections`

ExecutionTimeInSeconds

Get execution time in seconds for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds
```

Description

`ExecutionTimes = NthSectionProfile.ExecutionTimeInSeconds` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of `ExecutionTimes` contains the difference between the timer reading at the start and the end of the section.

Examples

Get Execution Times for Code Section

Copy MATLAB code to your working folder.

```
src_dir = ...
    fullfile(docroot, 'toolbox', 'coder', 'examples', 'kalman');

copyfile(fullfile(src_dir, 'kalman01.m'), '.')
copyfile(fullfile(src_dir, 'test01_ui.m'), '.')
copyfile(fullfile(src_dir, 'plot_trajectory.m'), '.')
copyfile(fullfile(src_dir, 'position.mat'), '.')
```

Set up and run a SIL execution.

```
config = coder.config('lib');
config.GenerateReport = true;

config.VerificationMode = 'SIL';
config.CodeExecutionProfiling = true;

codegen('-config', config, '-args', {zeros(2,1)}, 'kalman01');
```



```
coder.runTest('test01_ui', ['kalman01_sil.' mexext]);
```

At end of the execution, you see the following message.

```
To terminate execution: clear kalman01_sil
Execution profiling report available after termination.
```

Click the link `clear kalman01_sil`.

```
### Stopping SIL execution for 'kalman01'
    Execution profiling report: report(getCoderExecutionProfile('kalman01'))
```

Create a workspace variable that holds execution time data.

```
executionProfile=getCoderExecutionProfile('kalman01');
```

Get the profile for the second code section.

```
SecondSectionProfile = executionProfile.Sections(2);
```

Get vector of execution times for the code section.

```
time_vector = SecondSectionProfile.ExecutionTimeInSeconds;
```

Input Arguments

NthSectionProfile — `coder.profile.ExecutionTimeSection`

object

Object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

ExecutionTimes — Execution time measurements

double

Execution times, in seconds, for section of code. Returned as a vector.

More About

- “Generate Execution Time Profile”

- “Analyze Execution Time Data”

See Also

ExecutionTimeInTicks | getCoderExecutionProfile | Sections

ExecutionTimeInTicks

Get execution times in timer ticks for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Description

ExecutionTimes = *NthSectionProfile*.ExecutionTimeInTicks returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of *ExecutionTimes* contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

If you set the `CodeProfilingSaveOptions` parameter to `'SummaryOnly'`, *NthSectionProfile*.ExecutionTimeInTicks returns an empty array. To change that parameter, open the Configuration Parameters dialog by pressing **Ctrl+E**, open the **Verification** pane under **Code Generation**, and change the **Save options** parameter to **All measurement and analysis data**.

Tip You can calculate the execution time in seconds using the formula

$$ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond$$

Output Arguments

ExecutionTimes

Vector of execution times, in timer ticks, for profiled section of code

SelfExecutionTimes

Vector of execution times, in timer ticks, for profiled section of code but excluding time spent in child functions

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

ExecutionTimeInTicks

Get execution times in timer ticks for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.ExecutionTimeInTicks
```

Description

ExecutionTimes = *NthSectionProfile*.ExecutionTimeInTicks returns a vector of execution times, measured in timer ticks, for the profiled section of code. Each element of *ExecutionTimes* contains the difference between the timer reading at the start and the end of the section. The data type of the arrays is the same as the data type of the timer used on the target, which allows you to infer the maximum range of the timer measurements.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Tip You can calculate the execution time in seconds using the formula

$$ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond$$

Alternatively, `setTimerTicksPerSecond` and use `ExecutionTimeInSeconds`.

Output Arguments

ExecutionTimes

Vector of execution times, in timer ticks, for profiled section of code

More About

- “Generate Execution Time Profile”

- “Analyze Execution Time Data”

See Also

Sections | NumCalls | MaximumSelfTimeCallNum |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks
| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks | ExecutionTimeInSeconds |
getCoderExecutionProfile | TimerTicksPerSecond | report | Name | Number |
MaximumExecutionTimeCallNum | SelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum

MaximumExecutionTimeCallNum

Get the call number at which maximum number of timer ticks occurred

Syntax

```
MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum
```

Description

MaxTicksCallNum = *NthSectionProfile*.MaximumExecutionTimeCallNum returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during a simulation.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTicksCallNum

Call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `ExecutionTimeInTicks` | `Number` | `NumCalls` | `MaximumSelfTimeCallNum` | `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` |

TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

MaximumExecutionTimeCallNum

Get the call number at which maximum number of timer ticks occurred

Syntax

```
MaxTicksCallNum = NthSectionProfile.MaximumExecutionTimeCallNum
```

Description

MaxTicksCallNum = *NthSectionProfile*.MaximumExecutionTimeCallNum returns the call number at which the maximum number of timer ticks was recorded in a single invocation of the profiled code section during an execution.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTicksCallNum

Call number at which the maximum number of timer ticks occurred for a single invocation of the profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` |
`TotalExecutionTimeInTicks` | `MaximumSelfTimeInTicks` |
`TurnaroundTimeInTicks` | `TotalTurnaroundTimeInTicks` | `Sections` |

```
report | Name | Number | MaximumSelfTimeCallNum | SelfTimeInTicks  
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |  
MaximumTurnaroundTimeCallNum
```

MaximumExecutionTimeInTicks

Get maximum number of timer ticks for single invocation of profiled code section

Syntax

```
MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks
```

Description

MaxTicks = *NthSectionProfile*.MaximumExecutionTimeInTicks returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during a simulation.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

MaxTicks

Maximum number of timer ticks for single invocation of profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name |
ExecutionTimeInTicks | Number | NumCalls | MaximumExecutionTimeCallNum
| MaximumSelfTimeCallNum | ExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks

| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks
| MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

MaximumExecutionTimeInTicks

Get maximum number of timer ticks for single invocation of profiled code section

Syntax

```
MaxTicks = NthSectionProfile.MaximumExecutionTimeInTicks
```

Description

MaxTicks = *NthSectionProfile*.MaximumExecutionTimeInTicks returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during an execution.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTicks

Maximum number of timer ticks for single invocation of profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

```
getCoderExecutionProfile | TimerTicksPerSecond | NumCalls  
| MaximumSelfTimeCallNum | TotalExecutionTimeInTicks  
| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |  
TotalTurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks  
| MaximumTurnaroundTimeCallNum | Sections | report | Name |
```

Number | MaximumExecutionTimeCallNum | ExecutionTimeInTicks |
SelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks
| MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

TotalExecutionTimeInTicks

Get total number of timer ticks recorded for profiled code section

Syntax

```
TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks
```

Description

TotalTicks = *NthSectionProfile*.TotalExecutionTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire simulation.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

TotalTicks

Total number of timer ticks for profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks

| MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum |
TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

TotalExecutionTimeInTicks

Get total number of timer ticks recorded for profiled code section

Syntax

```
TotalTicks = NthSectionProfile.TotalExecutionTimeInTicks
```

Description

TotalTicks = *NthSectionProfile*.TotalExecutionTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire execution.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalTicks

Total number of timer ticks for profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls` | `MaximumSelfTimeCallNum` | `SelfTimeInTicks` | `TotalSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `Sections` | `report` | `Name` | `Number` | `MaximumExecutionTimeCallNum` | `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks`

| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

SelfTimeInTicks

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
SelfTicks = NthSectionProfile.SelfTimeInTicks
```

Description

SelfTicks = *NthSectionProfile*.SelfTimeInTicks returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

SelfTicks

Number of timer ticks for profiled code section, excluding periods in child functions

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name |
ExecutionTimeInTicks | Number | NumCalls | MaximumExecutionTimeCallNum
| MaximumSelfTimeCallNum | ExecutionTimeInTicks |

MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks
| MaximumSelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum |
TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

SelfTimeInTicks

Get number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
SelfTicks = NthSectionProfile.SelfTimeInTicks
```

Description

SelfTicks = *NthSectionProfile*.SelfTimeInTicks returns the number of timer ticks recorded for the profiled code section. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

SelfTicks

Number of timer ticks for profiled code section, excluding periods in child functions

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks`
| `MaximumSelfTimeInTicks` | `TurnaroundTimeInTicks` |
`TotalTurnaroundTimeInTicks` | `Sections` | `report` | `Name` |

Number | MaximumExecutionTimeCallNum | ExecutionTimeInTicks
| MaximumExecutionTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum

MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

Syntax

```
MaxSelfTicksCallNum = NthSectionProfile.MaxSelfTimeCallNum
```

Description

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxSelfTicksCallNum

Call number at which the maximum number of self-time ticks occurred for profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `ExecutionTimeInTicks` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum`

| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

MaximumSelfTimeCallNum

Get the call number at which the maximum number of timer ticks occurred, excluding time spent in child functions

Syntax

```
MaxSelfTicksCallNum = NthSectionProfile.MaxSelfTimeCallNum
```

Description

MaxSelfTicksCallNum = *NthSectionProfile*.MaxSelfTimeCallNum returns the call number at which the maximum number of self-time ticks occurred for the profiled code section.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxSelfTicksCallNum

Call number at which the maximum number of self-time ticks occurred for profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `ExecutionTimeInTicks` | `MaximumExecutionTimeInTicks` |
`TotalExecutionTimeInTicks` | `MaximumSelfTimeInTicks` |

TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks | Sections |
report | Name | Number | MaximumExecutionTimeCallNum | SelfTimeInTicks
| TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks |
MaximumTurnaroundTimeCallNum

MaximumSelfTimeInTicks

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks
```

Description

MaxSelfTicks = *NthSectionProfile*.MaximumSelfTimeInTicks returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxSelfTicks

Maximum number of timer ticks for profiled code section, excluding periods in child functions

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `ExecutionTimeInTicks` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum`

| MaximumSelfTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
SelfTimeInTicks | TotalSelfTimeInTicks | MaximumTurnaroundTimeInTicks
| MaximumTurnaroundTimeCallNum | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

MaximumSelfTimeInTicks

Get the maximum number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
MaxSelfTicks = NthSectionProfile.MaximumSelfTimeInTicks
```

Description

MaxSelfTicks = *NthSectionProfile*.MaximumSelfTimeInTicks returns the maximum number of timer ticks recorded for the profiled code section. This number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxSelfTicks

Maximum number of timer ticks for profiled code section, excluding periods in child functions

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks` |
`TotalSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` |

MaximumTurnaroundTimeCallNum | Sections | report | Name |
Number | MaximumExecutionTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | SelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

TotalSelfTimeInTicks

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks
```

Description

TotalSelfTicks = *NthSectionProfile*.TotalSelfTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire simulation. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalSelfTicks

Total number of timer ticks for profiled code section, excluding periods in child functions

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report`
| `Name` | `ExecutionTimeInTicks` | `Number` | `NumCalls` |
`MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum`

| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum |
TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

TotalSelfTimeInTicks

Get total number of timer ticks recorded for profiled code section, excluding time spent in child functions

Syntax

```
TotalSelfTicks = NthSectionProfile.TotalSelfTimeInTicks
```

Description

TotalSelfTicks = *NthSectionProfile*.TotalSelfTimeInTicks returns the total number of timer ticks recorded for the profiled code section over the entire execution. However, this number excludes the time spent in calls to child functions.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalSelfTicks

Total number of timer ticks for profiled code section, excluding periods in child functions

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls` | `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks` | `MaximumSelfTimeInTicks` | `MaximumTurnaroundTimeInTicks` | `MaximumTurnaroundTimeCallNum` | `Sections` | `report` | `Name` |

Number | MaximumExecutionTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | SelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks

MaximumTurnaroundTimeInTicks

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

```
MaxTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks
```

Description

MaxTicks = *NthSectionProfile*.MaximumTurnaroundTimeInTicks returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTurnaroundTicks

Maximum number of timer ticks between start and finish of a single invocation of profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum`

| MaximumTurnaroundTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks |
TurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

MaximumTurnaroundTimeInTicks

Get maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

```
MaxTicks = NthSectionProfile.MaximumTurnaroundTimeInTicks
```

Description

MaxTicks = *NthSectionProfile*.MaximumTurnaroundTimeInTicks returns the maximum number of timer ticks recorded between the start and finish of a single invocation of the profiled code section during a execution. Unless the code is pre-empted, this is the same as the maximum execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

MaxTurnaroundTicks

Maximum number of timer ticks between start and finish of a single invocation of profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks`

| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks | Sections | report | Name | Number
| MaximumExecutionTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeCallNum

MaximumTurnaroundTimeCallNum

Get call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

Syntax

```
MaxTurnaroundTicksCallNum =  
NthSectionProfile.MaximumTurnaroundTimeCallNum
```

Description

```
MaxTurnaroundTicksCallNum =  
NthSectionProfile.MaximumTurnaroundTimeCallNum
```

 returns the call number in which the maximum number of timer ticks was recorded between start and finish of a single invocation of the profiled code section during a simulation. Unless the code is pre-empted, this is the same as the maximum execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

```
MaxTurnaroundTicksCallNum
```

Call number of the maximum number of timer ticks between start and finish of a single invocation of profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number |
NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum
| ExecutionTimeInTicks | MaximumExecutionTimeInTicks |
TotalExecutionTimeInTicks | SelfTimeInTicks | MaximumSelfTimeInTicks
| TotalSelfTimeInTicks | TurnaroundTimeInTicks |
MaximumTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

MaximumTurnaroundTimeCallNum

Get call number for the code section invocation with the maximum number of timer ticks between the start and the finish

Syntax

```
MaxTurnaroundTicksCallNum =  
NthSectionProfile.MaximumTurnaroundTimeCallNum
```

Description

MaxTurnaroundTicksCallNum = *NthSectionProfile*.MaximumTurnaroundTimeCallNum returns the call number in which the maximum number of timer ticks is recorded between the start and the finish of an invocation of the profiled code section. Unless the code is pre-empted, this is the same as the maximum execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

MaxTurnaroundTicksCallNum

Call number for the profiled code section invocation with the maximum number of timer ticks between start and finish

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks`

| MaximumSelfTimeInTicks | TurnaroundTimeInTicks |
TotalTurnaroundTimeInTicks | Sections | report | Name | Number
| MaximumExecutionTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks

TotalTurnaroundTimeInTicks

Get total number of timer ticks between start and finish of the profiled code section over the entire simulation.

Syntax

```
TotalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks
```

Description

TotalTurnaroundTicks = *NthSectionProfile*.TotalTurnaroundTimeInTicks returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire simulation. Unless the code is pre-empted, this is the same as the total execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TotalTurnaroundTicks

Total number of timer ticks between start and finish of the profiled code section over the entire simulation

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

`Sections` | `TimerTicksPerSecond` | `display` | `report` | `Name` | `Number` | `NumCalls` | `MaximumExecutionTimeCallNum` | `MaximumSelfTimeCallNum`

| MaximumTurnaroundTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |
SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks |
TurnaroundTimeInTicks | MaximumTurnaroundTimeInTicks

TotalTurnaroundTimeInTicks

Get total number of timer ticks between start and finish of the profiled code section over the entire execution.

Syntax

```
totalTurnaroundTicks = NthSectionProfile.TotalTurnaroundTimeInTicks
```

Description

totalTurnaroundTicks = *NthSectionProfile*.TotalTurnaroundTimeInTicks returns the total number of timer ticks recorded between the start and finish of the profiled code section over the entire execution. Unless the code is pre-empted, this is the same as the total execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

totalTurnaroundTicks

Total number of timer ticks between start and finish of the profiled code section over the entire execution

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `TimerTicksPerSecond` | `NumCalls`
| `MaximumSelfTimeCallNum` | `TotalExecutionTimeInTicks` |

MaximumSelfTimeInTicks | TurnaroundTimeInTicks | Sections | report |
Name | Number | MaximumExecutionTimeCallNum | ExecutionTimeInTicks |
MaximumExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum

TurnaroundTimeInTicks

Get number of timer ticks between start and finish of the profiled code section

Syntax

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks
```

Description

TurnaroundTicks = *NthSectionProfile*.TurnaroundTimeInTicks returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property Sections.

Output Arguments

TurnaroundTicks

Number of timer ticks between start and finish of the profiled code section

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

See Also

Sections | TimerTicksPerSecond | display | report | Name | Number | NumCalls | MaximumExecutionTimeCallNum | MaximumSelfTimeCallNum | MaximumTurnaroundTimeCallNum | ExecutionTimeInTicks | MaximumExecutionTimeInTicks | TotalExecutionTimeInTicks |

SelfTimeInTicks | MaximumSelfTimeInTicks | TotalSelfTimeInTicks |
MaximumTurnaroundTimeInTicks | TotalTurnaroundTimeInTicks

TurnaroundTimeInTicks

Get number of timer ticks between start and finish of the profiled code section

Syntax

```
TurnaroundTicks = NthSectionProfile.TurnaroundTimeInTicks
```

Description

TurnaroundTicks = *NthSectionProfile*.TurnaroundTimeInTicks returns the number of timer ticks recorded between the start and finish of the profiled code section. Unless the code is pre-empted, this is the same as the execution time.

NthSectionProfile is a `coder.profile.ExecutionTimeSection` object generated by the `coder.profile.ExecutionTime` property `Sections`.

Output Arguments

TurnaroundTicks

Number of timer ticks between start and finish of the profiled code section

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

```
getCoderExecutionProfile | TimerTicksPerSecond | NumCalls  
| MaximumSelfTimeCallNum | TotalExecutionTimeInTicks |  
MaximumSelfTimeInTicks | Sections | report | Name | Number  
| MaximumExecutionTimeCallNum | ExecutionTimeInTicks |  
MaximumExecutionTimeInTicks | SelfTimeInTicks | TotalSelfTimeInTicks
```

| MaximumTurnaroundTimeInTicks | MaximumTurnaroundTimeCallNum |
TotalTurnaroundTimeInTicks

modifyInheritedParam

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Modify inherited parameter values

Syntax

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

Description

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

Input Arguments

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

Examples

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

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

See Also

`get_param`

How To

- “Create Custom Objectives”
- “Parameter Command-Line Information Summary”

msgcount

Number of messages in read-enabled channel queue

Note: Support for msgcount on C5000 processors will be removed in a future version.

Syntax

```
msgcount(rx, 'channel')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`msgcount(rx, 'channel')` returns the number of unread messages in the read-enabled queue specified by `channel` for the RTDX interface `rx`. You cannot use `msgcount` on channels configured for write access.

Examples

If you have created and loaded a program to the processor, you can write data to the processor, then use `msgcount` to determine the number of messages in the read queue.

- 1 Create and load a program to the processor.
- 2 Write data to the processor from MATLAB software.

```
indata = 1:100;  
writemsg(rtdx(IDE_Obj), 'ichannel', int32(indata));
```

- 3 Use `msgcount` to determine the number of messages available in the queue.

```
num_of_msgs = msgcount(rtdx(IDE_Obj), 'ichannel')
```

See Also

`read` | `readmat` | `readmsg`

new

Create project, library, or build configuration in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

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

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

The new method does not support 'text' as a *type* argument.

Examples

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

See Also

activate | close

open

Open project in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

	CCS IDE	VisualDSP++ IDE
'project' — Project files	Yes	Yes
'ProjectGroup' — Project group files	No	Yes
'program' — Target program file (executable)	No. Use <code>load</code> instead.	No

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

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

Note: The open method does not support the 'text', 'program', or 'workspace' arguments.

Examples

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

See Also

`"cd" | dir | load | new`

plot

Class: `cgv.CGV`

Package: `cgv`

Create plot for signal or multiple signals

Syntax

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

Description

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

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

Input Arguments

`data_set`

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

`'Signals', signal_list`

Parameter/value argument pair specifying the signal or signals to plot. The value for this parameter can be an individual signal name, or a cell array of strings, where each string is a signal name in the `data_set`. Use “`getSavedSignals (cgv.CGV)`” to view the list of available signal names in the `data_set`. The syntax for an individual signal name is:

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

The syntax for a list of signal names is:

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

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

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

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

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

Output Arguments

Depending on the data, one or more of the following parameters might be empty:

signal_names

Cell array of signal names

signal_figures

Array of figure handles for signals

How To

- “Verify Numerical Equivalence with CGV”

profile

Generate real-time execution or stack profiling report

Syntax

```
profile(IDE_Obj,type,action,timeout)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

Use `profile(IDE_Obj,type,action,timeout)` to generate real-time execution or stack profiling report.

Create the *IDE_Obj* IDE handle object using a constructor function before you use the `profile` method.

The *type* argument determines the type of profile to generate. The following types are available for the IDEs specified.

	CCS IDE	VisualDSP++ IDE
'execution' — Execution profiling	Yes	Yes
'stack' — Stack profiling	Yes	Yes

To get a real-time task execution profile report in HTML and graphical plot forms, set the *type* argument to 'execution' and omit the *action* argument, which defaults to

'report'. For more information, see “Perform Execution Time Profiling for IDE and Toolchain Targets”.

To prepare the stack memory on the processor for profiling, set the *type* argument to 'stack', and set the *action* argument to 'setup'. This action writes a repetitive series of known values to the stack memory. For more information, see “Perform Stack Profiling with IDE and Toolchain Targets”.

After preparing the stack memory, to measure and report the percentage of stack usage, set the *type* argument to 'stack', and set the *action* argument to 'report'.

If you omit the *action* argument, *action* defaults to 'report'.

The optional *timeout* argument determines the number of seconds MATLAB waits for the IDE to finish profiling before returning an error. If you omit the *timeout* argument, the open method uses the timeout property of the IDE handle object (IDE_Obj) instead.

Note: You can use real-time task execution profiling with hardware only. Simulators do not support the profiling feature.

Examples

To use `profile` to assess how your program executes in real-time, complete the following tasks with a Simulink model:

- 1 Open the model configuration parameters (**Ctrl+ E**).
- 2 Select the Coder Target pane.
- 3 Under the Tool Chain Automation tab, enable **Profile real-time execution**.
- 4 Build your model.

```
build(IDE_Obj)
```

- 5 Load your program to the processor.

```
load(IDE_Obj, 'c:\work\sumdiff.out')
```

- 6 For stack profiling, initialize the stack to a known state. (For execution profiling, skip this step.)

```
profile(IDE_Obj, 'stack', 'setup')
```

With the **setup** input argument, **profile** writes a known pattern into the addresses that compose the stack. For C6000 processors, the pattern is A5. For C2000™ and C5000 processors, the pattern is A5A5 to account for the address size. As long as your application does not write the same pattern to the system stack, **profile** can report the stack usage.

- 7 Run the program on the processor.

```
run(IDE_Obj)
```

- 8 Stop the running program.

```
halt(IDE_Obj)
```

- 9 To get the profiling reports enter one of the following commands:

```
profile(IDE_Obj,'stack','report') #Get stack profiling report
profile(IDE_Obj,'execution') #Get execution profiling report
```

The HTML report contains the sections described in the following table.

Section Heading	Description
Worst case task turnaround times	Maximum task turnaround time for each task since model execution started.
Maximum number of concurrent overruns for each task	Maximum number of concurrent task overruns since model execution started.
Analysis of profiling data recorded over <i>nnn</i> seconds.	Profiling data was recorded over <i>nnn</i> seconds. The recorded data for task turnaround times and task execution times is presented in the table following this heading.

Task turnaround time is the elapsed time between starting and finishing the task. If the task is not preempted, task turnaround time equals the task execution time.

Task execution time is the time between task start and finish when the task is actually running. It does not include time during which the task may have been preempted by another task.

Note: Task execution time cannot be measured directly. Task profiling infers the execution time from the task start and finish times, and the intervening periods during which the task was preempted by another task.

The execution time calculations do not account for processor time consumed by the scheduler while switching tasks. In cases where preemption occurs, the reported task execution times overestimate the true task execution time.

Task overruns occur when a timer task does not complete before the same task is scheduled to run again. Depending on how you configure the real-time scheduler, a task overrun may be handled as a real-time failure. Alternatively, you might allow a small number of task overruns to accommodate cases where a task occasionally takes longer than normal to complete. If a task overrun occurs, and the same task is scheduled to run again before the first overrun has been cleared, concurrent task overruns are said to have occurred.

See Also

load | run

read

Read data from processor memory

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

- The start address
- The memory type

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

Alternatively, the *IDE_Obj* object has a default memory type value that is applied if the memory type value is not explicitly incorporated in the passed address parameter. In

DSP processors with only a single memory type, it is possible to specify addresses using the abbreviated (implied memory type) format by setting the `IDE_Obj` object memory type value to zero.

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

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

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

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

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

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

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

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

`mem = read(IDE_Obj, ..., datatype)` where the input argument `datatype` defines the interpretation of the raw values read from DSP memory. Parameter `datatype` specifies

the data format of the raw memory image. The data is read starting from **address** without regard to data type alignment boundaries in the processor. The byte ordering defined by the data type is automatically applied. This syntax supports the following MATLAB data types.

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

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

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

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

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

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

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

Examples

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

```
m1var = read(IDE_Obj, 131072, 'int16')  
131072 is the decimal address of the data to read.
```

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

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

See Also

`write`

readmat

Matrix of data from RTDX channel

Note: Support for `readmat` on C5000 processors will be removed in a future version.

Syntax

```
data = readmat(rx,channelname,'datatype',siz,timeout)
data = readmat(rx,channelname,'datatype',siz)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`data = readmat(rx,channelname,'datatype',siz,timeout)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Before you read from a channel, open and enable the channel for read access.

Replace `channelname` with the string you specified when you opened the desired channel. `channelname` must identify a channel that you defined in the program loaded on the processor.

You cannot read data from a channel you have not opened and configured for read access. To determine which channels exist for the loaded program, use the RTDX tools provided in the IDE.

`data` contains a matrix whose dimensions are given by the input argument vector `siz`, where `siz` can be a vector of two or more elements. To operate, the number of elements in the output matrix `data` must be an integral number of channel messages.

When you omit the `timeout` input argument, `readmat` reads messages from the specified channel until the output matrix is full or the global `timeout` period specified in `rx` elapses.

Caution If the timeout period expires before the output data matrix is fully populated, you lose the messages read from the channel to that point.

MATLAB software supports reading five data types with `readmat`.

datatype String	Data Format
'double'	Double-precision floating point values. 64 bits.
'int16'	16-bit signed integers
'int32'	32-bit signed integers
'single'	Single-precision floating point values. 32 bits.
'uint8'	Unsigned 8-bit integers

`data = readmat(rx,channelname,'datatype',siz)` reads a matrix of data from an RTDX channel configured for read access. `datatype` defines the type of data to read, and `channelname` specifies the queue to read. `readmat` reads the desired data from the RTDX link specified by `rx`.

Examples

In this data read and write example, you write data to the processor through the IDE. You can then read the data back in two ways — either through `read` or through `readmsg`.

To duplicate this example you need to have a program loaded on the processor. The channels listed in this example, `ichannel` and `ochannel`, must be defined in the loaded program. If the current program on the processor defines different channels, replace the listed channels with your current ones.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(IDE_Obj,0,indata,30);
outdata = read(IDE_Obj,0,'double',25,10)

outdata =
  Columns 1 through 13
   1   2   3   4   5   6   7   8   9  10  11  12  13
  Columns 14 through 25
  14  15  16  17  18  19  20  21  22  23  24  25
```

Now use RTDX to read the data into a 5-by-5 array called `out_array`.

```
out_array = readmat('ochannel','double',[5 5])
```

See Also

[readmsg](#) | [writemsg](#)

readmsg

Read messages from specified RTDX channel

Note: Support for readmsg on C5000 processors will be removed in a future version.

Syntax

```
data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)
data = readmsg(rx,channelname,'datatype',siz,nummsgs)
data = readmsg(rx,channelname,datatype,siz)
data = readmsg(rx,channelname,datatype,nummsgs)
data = readmsg(rx,channelname,datatype)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`data = readmsg(rx,channelname,'datatype',siz,nummsgs,timeout)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read the messages present in the queue when you call the `readmsg` function.

Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. For example, when `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `m`-by-`n` matrices in `data`. Each output matrix in `data` must have the same number of elements (`m x n`) as the number of elements in each message.

You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports strings that define the type of data you are expecting, as shown in the following table.

datatype String	Specified Data Type
'double'	Floating point data, 64-bits (double-precision).
'int16'	Signed 16-bit integer data.
'int32'	Signed 32-bit integers.
'single'	Floating-point data, 32-bits (single-precision).
'uint8'	Unsigned 8-bit integers.

When you include the `timeout` input argument in the function, `readmsg` reads messages from the specified queue until it receives `nummsgs`, or until the period defined by `timeout` expires while `readmsg` waits for more messages to be available.

When the desired number of messages is not available in the queue, `readmsg` enters a wait loop and stays there until more messages become available or `timeout` seconds elapse. The `timeout` argument overrides the global timeout specified when you create `rx`.

`data = readmsg(rx,channelname,'datatype',siz,nummsgs)` reads `nummsgs` from a channel associated with `rx`. `channelname` identifies the channel queue, which must be configured for read access. Each message is the same type, defined by `datatype`. `nummsgs` can be an integer that defines the number of messages to read from the specified queue, or `all` to read the messages present in the queue when you call the `readmsg` function.

Each read message becomes an output matrix in `data`, with dimensions specified by the elements in vector `siz`. When `siz` is `[m n]`, reading 10 messages (`nummsgs` equal 10) creates 10 `n`-by-`m` matrices in `data`.

Each output matrix in `data` must have the same number of elements (`m x n`) as the number of elements in each message.

You must specify the type of messages you are reading by including the `datatype` argument. `datatype` supports six strings that define the type of data you are expecting.

`data = readmsg(rx,channelname,datatype,siz)` reads one data message because `nummsgs` defaults to one when you omit the input argument. `readmsg` returns the message as a row vector in `data`.

`data = readmsg(rx,channelname,datatype,nummsgs)` reads the number of messages defined by `nummsgs`. `data` becomes a cell array of row matrices, `data = {msg1,msg2,...,msg(nummsgs)}`, because `siz` defaults to `[1,nummsgs]`; each returned message becomes one row matrix in the cell array.

Each row matrix contains one element for each data value in the current message `msg# = [element(1), element(2), ..., element(l)]` where `l` is the number of data elements in message. In this syntax, the read messages can have different lengths, unlike the previous syntax options.

`data = readmsg(rx,channelname,datatype)` reads one data message, returning a row vector in `data`. The optional input arguments—`nummsgs`, `siz`, and `timeout`—use their default values.

In the calling syntaxes for `readmsg`, you can set `siz` and `nummsgs` to empty matrices, causing them to use their default values—`nummsgs = 1` and `siz = [1,l]`, where `l` is the number of data elements in the read message.

Caution If the timeout period expires before the output data matrix is fully populated, you lose the messages read from the channel to that point.

Examples

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
write(IDE_Obj,0,indata,30);
outdata = read(IDE_Obj,0,'double',25,10)

outdata =
  Columns 1 through 13
   1  2  3  4  5  6  7  8  9 10 11 12 13
  Columns 14 through 25
 14 15 16 17 18 19 20 21 22 23 24 25
```

Now use `RTDX` to read the messages into a 4-by-5 array called `out_array`.

```
number_msgs = msgcount(rx,'ochannel') % Check number of msgs
% in read queue.
out_array = rtdx(IDE_Obj).readmsg('ochannel','double',[4 5])
```

See Also

read | readmat | writemsg

register

Class: `rtw.codegenObjectives.Objective`

Package: `rtw.codegenObjectives`

Register objective

Syntax

```
register(obj)
```

Description

`register(obj)` registers *obj* Register and add *obj* to the end of the list of available objectives that you can use with the Code Generation Advisor.

Input Arguments

obj Handle to a code generation objective object previously created.

Examples

Register the objective:

```
register(obj);
```

See Also

How To

- “Create Custom Objectives”
- “Registering Customizations”

registerCFunctionEntry

Create function entry based on specified parameters and register in code replacement table

Syntax

```
entry = registerCFunctionEntry(hTable, priority,  
                               numInputs, functionName,  
                               inputType, implementationName,  
                               outputType, headerFile,  
                               genCallback, genFileName)
```

Input Arguments

hTable

Handle to a code replacement table previously returned by *hTable* = RTW.Tf1Table.

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 a function listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

inputType

String specifying the data type of the input arguments, for example, 'double'. (This function requires that the 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 “Specify Build Information for Replacement Code”.

genFileName

String specifying ''. (This argument is for use only by MathWorks developers.)

Output Arguments

Handle to the created code replacement function entry. Specifying the return argument in the registerCFunctionEntry function call is optional.

Description

The registerCFunctionEntry function provides a quick way to create and register a code replacement function entry. This function can be used only if your function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:
 - For input argument names, *u1*, *u2*, ..., *un*
 - For return argument, *y1*

Examples

In the following example, the `registerCFunctionEntry` function is used to create a function entry for `sqrt` in a code replacement table.

```
hLib = RTW.Tf1Table;  
hLib.registerCFunctionEntry(100, 1, 'sqrt', 'double', 'sqrt', ...  
                           'double', '<math.h>', '', '');
```

More About

- “Define Code Replacement Mappings”

See Also

`registerCPromotableMacroEntry`

registerCPPFunctionEntry

Create C++ function entry based on specified parameters and register in code replacement table

Syntax

```
entry = registerCPPFunctionEntry(hTable, priority,  
                                numInputs, functionName,  
                                inputType, implementationName,  
                                outputType, headerFile,  
                                genCallback, genFileName,  
                                nameSpace)
```

Input Arguments

hTable

Handle to a code replacement 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 a function listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

inputType

String specifying the data type of the input arguments, for example, 'double'. (This function requires that the 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 “Specify Build Information for Replacement Code”.

genFileName

String specifying ''. (This argument is for use only by MathWorks developers.)

nameSpace

String specifying the C++ namespace in which the implementation function is defined. If this function entry is matched, the software emits the namespace in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`). If you specify '', the software does not emit a namespace designation in the generated code.

Output Arguments

Handle to the created C++ function entry. Specifying the return argument in the registerCPPFunctionEntry function call is optional.

Description

The registerCPPFunctionEntry function provides a quick way to create and register a code replacement C++ function entry. This function can be used only if your C++ function entry meets the following conditions:

- The input arguments are of the same type.

- The input argument names and the return argument name follow the default Simulink naming convention:
 - For input argument names, *u1*, *u2*, ..., *un*
 - For return argument, *y1*

Note: When you register a code replacement library containing C++ function entries, you must specify the value { 'C++ ' } for the `LanguageConstraint` property of the library registry entry. For more information, see “Register Code Replacement Mappings”.

Examples

In the following example, the `registerCPPFunctionEntry` function is used to create a C++ function entry for `sin` in a code replacement table.

```
hLib = RTW.Tf1Table;  
hLib.registerCPPFunctionEntry(100, 1, 'sin', 'single', 'sin', ...  
                             'single', 'cmath', '', '', 'std');
```

More About

- “Define Code Replacement Mappings”

See Also

`enableCPP` | `setNameSpace`

registerCPromotableMacroEntry

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

Syntax

```
entry = registerCPromotableMacroEntry(hTable, priority,  
                                     numInputs, functionName,  
                                     inputType, implementationName,  
                                     outputType, headerFile,  
                                     genCallback, genFileName)
```

Input Arguments

hTable

Handle to a code replacement 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 the 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 “Specify Build Information for Replacement Code”.

genFileName

String specifying ''. (This argument is for use only by MathWorks developers.)

Output Arguments

Handle to the created promotable macro entry. Specifying the return argument in the registerCPromotableMacroEntry function call is optional.

Description

The registerCPromotableMacroEntry function creates a promotable macro entry based on specified parameters and registers the entry in the code replacement 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 promotable macro entry. This function can be used only if your code replacement function entry meets the following conditions:

- The input arguments are of the same type.
- The input argument names and the return argument name follow the default Simulink naming convention:

- For input argument names, u_1, u_2, \dots, u_n
- For return argument, y_1

Note: This function should be used only for **abs** function replacement. Other functions supported for replacement should use **registerCFunctionEntry**.

Examples

In the following example, the **registerCPromotableMacroEntry** function is used to create a function entry for **abs** in a code replacement table.

```
hLib = RTW.Tf1Table;  
hLib.registerCPromotableMacroEntry(100, 1, 'abs', 'double', 'abs_prime', ...  
    'double', '<math_prime.h>', '', '');
```

More About

- “Define Code Replacement Mappings”

See Also

registerCFunctionEntry

regread

Values from processor registers

Syntax

```
reg = regread(IDE_Obj, 'regname', 'represent', timeout)
reg = regread(IDE_Obj, 'regname', 'represent')
reg = regread(IDE_Obj, 'regname')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`reg = regread(IDE_Obj, 'regname', 'represent', timeout)` reads the data value in the `regname` register of the target processor and returns the value in `reg` as a double-precision value. For convenience, `regread` converts each return value to the MATLAB double datatype. Making this conversion lets you manipulate the data in MATLAB. String `regname` specifies the name of the source register on the target. The IDE handle, `IDE_Obj`, defines the target to read from. Valid entries for `regname` depend on your target processor.

Note: `regread` does not read 64-bit registers, like the `cycle` register on Blackfin processors.

Register names are not case-sensitive — `a0` is the same as `A0`.

For example, MPC5500 processors provide the following register names that are valid entries for `regname`.

Register Names	Register Contents
'acc'	Accumulator A register
sprg0 through sprg7	SPR registers

For example, TMS320C6xxx processors provide the following register names that are valid entries for `regname`.

Register Names	Register Contents
A0, A1, A2, ..., A15	General purpose A registers
B0, B1, B2, ..., B15	General purpose B registers
PC, ISTEP, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1, ..., B15:B14	64-bit general purpose register pairs

Note: Use `read` (called a direct memory read) to read memory-mapped registers.

The `represent` input argument defines the format of the data stored in `regname`. Input argument `represent` takes one of three input strings.

represent String	Description
'2scomp'	Source register contains a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
'binary'	Source register contains an unsigned binary integer.
'ieee'	Source register contains a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are reading from 32 and 64 bit registers on the target.

To limit the time that `regread` spends transferring data from the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the `timeout` argument, `regread` defaults to the global time-out defined in `IDE_Obj`.

`reg = regread(IDE_Obj, 'regname', 'represent')` does not set the global time-out value. The time-out value in `IDE_Obj` applies.

`reg = regread(IDE_Obj, 'regname')` does not define the format of the data in `regname`.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables and local variables during intermediate times in program operation may not get reflected in the register.

To see if the result is consistent, write a line of code that uses the variable. For example:

```
register int a = 100;
int b;
...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or the software.

Examples

For CCS IDE

For the C5xxx processor family, most registers are memory-mapped and consequently are available using `read` and `write`. However, use `regread` to read the PC register. The following command shows how to read the PC register. To identify the processor, `IDE_Obj` is a link for CCS IDE.

```
regread(IDE_Obj, 'PC', 'binary')
```


To tell MATLAB software what datatype you are reading, the string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

In response, MATLAB software displays

```
ans =  
  
    33824
```

For processors in the C6xxx family, `regread` lets you access processor registers directly. To read the value in general purpose register A0, type the following function.

```
treg = regread(IDE_Obj, 'A0', '2scomp');
```

`treg` now contains the two's complement representation of the value in A0.

Now read the value stored in register B2 as an unsigned binary integer, by typing

```
regread(IDE_Obj, 'B2', 'binary');
```

See Also

`read` | `regwrite` | `write`

regwrite

Write data values to registers on processor

Syntax

```
regwrite(IDE_Obj, 'regname', value, 'represent', timeout)
regwrite(IDE_Obj, 'regname', value, 'represent')
regwrite(IDE_Obj, 'regname', value,)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`regwrite(IDE_Obj, 'regname', value, 'represent', timeout)` writes the data in `value` to the `regname` register of the target processor. `regwrite` converts `value` from its representation in the MATLAB workspace to the representation specified by `represent`. The `represent` input argument defines the format of the data when it is stored in `regname`. Input argument `represent` takes one of three input strings.

represent String	Description
'2scomp'	Write <code>value</code> to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the <code>represent</code> argument.
'binary'	Write <code>value</code> to the destination register as an unsigned binary integer.
'ieee'	Write <code>value</code> to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the target.

Note: Use `write` to write memory-mapped registers. This action is also called a *direct memory write*.

String `regname` specifies the name of the destination register on the target. IDE handle, `IDE_Obj` defines the target to write `value` to. Valid entries for `regname` depend on your target processor. Register names are not case-sensitive — `a0` is the same as `A0`.

For example, MPC5500 processors provide the following register names that are valid entries for `regname`.

Register Names	Register Contents
'acc'	Accumulator A register
sprg0	SPR registers

For example, C6xxx processors provide the following register names that are valid entries for `regname`.

Register Names	Register Contents
A0, A1, A2,..., A15	General purpose A registers
B0, B1, B2,..., B15	General purpose B registers
PC, ISTEP, IFR, IRP, NRP, AMR, CSR	Other general purpose 32-bit registers
A1:A0, A2:A1,..., B15:B14	64-bit general purpose register pairs

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

To limit the time that `regwrite` spends transferring data to the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You might find this useful for limiting prolonged data transfer operations.

If you omit the `timeout` input argument in the syntax, `regwrite` defaults to the global time-out defined in `IDE_Obj`. If the write operation exceeds the time specified, `regwrite` returns with a time-out error. Generally, time-out errors do not stop the register write process. The write process stops while waiting for the IDE to respond that the write operation is complete.

`regwrite(IDE_Obj, 'regname', value, 'represent')` omits the `timeout` input argument and does not change the time-out value specified in `IDE_Obj`.

`regwrite(IDE_Obj, 'regname', value,)` omits the `represent` input argument. Writing the data does not reformat the data written to `regname`.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables and local variables during intermediate times in program operation may not get reflected in the register.

To see if the result is consistent, write a line of code that uses the variable. For example:

```
register int a = 100;
int b;
...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or the software.

Examples

To write a new value to the PC register on a C5xxx family processor, enter

```
regwrite(IDE_Obj, 'pc', hex2dec('100'), 'binary')
```

specifying that you are writing the value 256 (the decimal value of 0x100) to register `pc` as binary data.

To write a 64-bit value to a register pair, such as B1:B0, the following syntax specifies the value as a string, representation, and target registers.

```
rewrite(IDE_Obj, 'b1:b0', hex2dec('1010'), 'ieee')
```

Registers B1:B0 now contain the value 4112 in double-precision format.

See Also

read | regread | write

reload

Reload most recent program file to processor signal processor

Syntax

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

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

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

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

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

Using reload with Multiprocessor Boards

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

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

Examples

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

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

s =
    ''

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

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

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

See Also

“`cd`” | `load` | `open`

remove

Remove file, project, or breakpoint

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

Input Arguments

IDE_Obj

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

filename

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

filetype

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

Default: 'text'

addr

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

debugtype

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

Default: 'break' (breakpoint)

line

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

timeout

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

Examples

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

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

Now remove one file from your project

```
remove(IDE_Obj, 'filename')
```

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

See Also

add | “cd” | open

removeInheritedCheck

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Remove inherited checks

Syntax

```
removeInheritedCheck(obj, checkID)
```

Description

`removeInheritedCheck(obj, checkID)` removes an inherited check from the objective definition. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

Input Arguments

<i>obj</i>	Handle to a code generation objective object previously created.
<i>checkID</i>	Unique identifier of the check that you remove from the new objective.

Examples

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

```
removeInheritedCheck(obj, 'mathworks.codegen.CodeInstrumentation');  
)
```

See Also

Simulink.ModelAdvisor

How To

- “Create Custom Objectives”
- “About IDs”

removeInheritedParam

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Remove inherited parameters

Syntax

```
removeInheritedParam(obj, paramName)
```

Description

`removeInheritedParam(obj, paramName)` removes an inherited parameter from this objective. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another objective includes the parameter, the Code Generation Advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

Input Arguments

<i>obj</i>	Handle to a code generation objective object previously created.
<i>paramName</i>	Parameter that you want to remove from the objective.

Examples

Remove `InlineParameters` from the objective.

```
removeInheritedParam(obj, 'InlineParams');
```

See Also

`get_param`

How To

- “Create Custom Objectives”
- “Parameter Command-Line Information Summary”

report

Open code execution profiling report and specify display of time measurements.

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)
report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor',
'1e-06', 'NumericFormat', '%0.3f')
```

Description

When you run a SIL or PIL simulation with code execution profiling, the software generates the workspace variable *myExecutionProfile*, specified in **Configuration Parameters > Code Generation > Verification > Workspace variable**.

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)` opens the report with display options specified by the name-value *string* pairs.

`report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')` displays time in microseconds (10^{-6} seconds) with a precision of three decimal places.

Name-Value Pair	Details
'Units', 'Seconds' or 'Units', 'Ticks'	<p>Time measurements displayed in seconds or timer ticks.</p> <p>Default:</p> <ul style="list-style-type: none"> • SIL simulation on Windows — Seconds • SIL simulation on non-Windows — Timer ticks • PIL simulation — Seconds, if number of timer ticks per second has been specified by the target connectivity configuration. Otherwise, ticks.

Name-Value Pair	Details
'ScaleFactor', <i>Value</i>	Scale factor for displayed measurements. For example, to display measurements in microseconds, use the name-value pair 'ScaleFactor', '1e-6'. <i>Value</i> must be a string representation of a number that is a power of 10. For example, '1', '1e-6', or '1e-9'. Default value is '1e-9'. To specify the scale factor, you must also specify 'Units', 'Seconds'.
'NumericFormat', <i>Convention</i>	Numeric format for displayed measurements. Use the <i>decimal</i> convention utilized by the ANSI [®] C function <code>sprintf</code> , for example, '%1.2f'. Default is '%0.0f'. To specify the numeric format, you must also specify 'Units', 'Seconds'.

More About

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”

See Also

display

report

Open code execution profiling report and specify display of time measurements.

Syntax

```
report(myExecutionProfile)
report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)
report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor',
'1e-06', 'NumericFormat', '%0.3f')
```

Description

`report(myExecutionProfile)` opens the code execution profiling report using default display options.

`report(myExecutionProfile, Name1, Value1, Name2, Value2, ...)` opens the report with display options specified by the name-value *string* pairs.

`report(myExecutionProfile, 'Units', 'Seconds', 'ScaleFactor', '1e-06', 'NumericFormat', '%0.3f')` displays time in microseconds (10^{-6} seconds) with a precision of three decimal places.

`myExecutionProfile` is a workspace variable that you create using `getCoderExecutionProfile`.

Name-Value Pair	Details
'Units', 'Seconds' or 'Units', 'Ticks'	Time measurements displayed in seconds or timer ticks. Default: <ul style="list-style-type: none"> • SIL execution on Windows — Seconds • SIL execution on non-Windows — Timer ticks • PIL execution — Seconds, if number of timer ticks per second has been specified by the target connectivity configuration. Otherwise, ticks.

Name-Value Pair	Details
'ScaleFactor', <i>Value</i>	Scale factor for displayed measurements. For example, to display measurements in microseconds, use the name-value pair 'ScaleFactor', '1e-6'. <i>Value</i> must be a string representation of a number that is a power of 10. For example, '1', '1e-6', or '1e-9'. Default value is '1e-9'. To specify the scale factor, you must also specify 'Units', 'Seconds'.
'NumericFormat', <i>Convention</i>	Numeric format for displayed measurements. Use the <i>decimal</i> convention utilized by the ANSI C function <code>sprintf</code> , for example, '%1.2f'. Default is '%0.0f'. To specify the numeric format, you must also specify 'Units', 'Seconds'.

More About

- “Generate Execution Time Profile”
- “Analyze Execution Time Data”

See Also

`getCoderExecutionProfile` | `Sections` | `TimerTicksPerSecond`

reset

Stop program execution and reset processor

Syntax

```
reset(IDE_Obj,timeout)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

`reset(IDE_Obj,timeout)` stops the program executing on the processor and asynchronously performs a processor reset, returning the processor register contents to their power-up settings. `reset` returns immediately after the processor halt.

The optional *timeout* argument sets the number of seconds MATLAB waits for the processor to halt. If you omit the timeout argument, timeout defaults to the timeout value of the IDE handle object.

See Also

halt | load | run

restart

Reload most recent program file to processor signal processor

Syntax

```
restart(IDE_Obj)  
restart(IDE_Obj,timeout)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

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

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

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

See Also

halt | isrunning | run

rtIOStreamClose

Shut down communications channel with remote processor

Syntax

```
int rtIOStreamClose(int streamID)
```

Arguments

streamID

A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

Description

`int rtIOStreamClose(int streamID)` shuts down the communications channel and cleans up associated resources.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_rtiostream_script`
- `rtwdemo_custom_pil_script`

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtiostream_wrapper`

rtIOStreamClose

Shut down communications channel with remote processor

Syntax

```
int rtIOStreamClose(int streamID)
```

Arguments

streamID

A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

Description

`int rtIOStreamClose(int streamID)` shuts down the communications channel and cleans up associated resources.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtIOStreamRecv` | `rtiostream_wrapper` | `rtIOStreamOpen` | `rtIOStreamSend`

rtIOStreamOpen

Initialize communications channel with remote processor

Syntax

```
int rtIOStreamOpen(int argc, void * argv[ ])
```

Arguments

argc

Integer argument count, i.e., the number of parameters in *argv[]*

argv[]

An array of pointers to parameters; typically these are null-terminated string parameters, however, this is allowed to be implementation dependent.

Description

`int rtIOStreamOpen(int argc, void * argv[])` initializes a communication stream to allow exchange of data between host and target.

The input parameters allows driver-specific parameters to be passed to the communications driver.

If able to initialize a communication stream, the function returns a nonnegative integer greater than zero, representing a stream handle. A return value of `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

More About

- “Create PIL Target Connectivity Configuration”

- `rtwdemo_rtiostream_script`
- `rtwdemo_custom_pil_script`

See Also

`rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose` | `rtiostream_wrapper`

rtIOStreamOpen

Initialize communications channel with remote processor

Syntax

```
int rtIOStreamOpen(int argc, void * argv[ ])
```

Arguments

argc

Integer argument count, i.e., the number of parameters in `argv[]`

argv[]

An array of pointers to parameters; typically these are null-terminated string parameters, however, this is allowed to be implementation dependent.

Description

`int rtIOStreamOpen(int argc, void * argv[])` initializes a communication stream to allow exchange of data between host and target.

The input parameters allows driver-specific parameters to be passed to the communications driver.

If able to initialize a communication stream, the function returns a nonnegative integer greater than zero, representing a stream handle. A return value of `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

More About

- “Create PIL Target Connectivity Configuration”

See Also

rtIOStreamRecv | rtiostream_wrapper | rtIOStreamClose | rtIOStreamSend

rtIOStreamRecv

Receive data from remote processor

Syntax

```
int rtIOStreamRecv(int streamID, void * dst, size_t size, size_t *
sizeRecvd)
```

Arguments

streamID

A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

size

Size of data to copy into the buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

dst

A pointer to the start of the buffer where received data must be copied.

sizeRecvd

The number of units of data received and copied into the buffer `dst` (zero if data was not copied).

Description

```
int rtIOStreamRecv(int streamID, void * dst, size_t size, size_t *
sizeRecvd)
```

 receives data from a remote processor through a communication channel.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See also `rtiostreamSend` for implementation and performance considerations.

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_rtiostream_script`
- `rtwdemo_custom_pil_script`

See Also

`rtIOStreamSend` | `rtIOStreamOpen` | `rtIOStreamClose` | `rtIOStream_wrapper`

rtIOStreamRecv

Receive data from remote processor

Syntax

```
int rtIOStreamRecv(int streamID, void * dst, size_t size, size_t * sizeRecvd)
```

Arguments

streamID

A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

size

Size of data to copy into the buffer. For byte-addressable architectures, `size` is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, `size` is measured in number of WORDs, where `sizeof(WORD) == 1`.

dst

A pointer to the start of the buffer where received data must be copied.

sizeRecvd

The number of units of data received and copied into the buffer `dst` (zero if data was not copied).

Description

```
int rtIOStreamRecv(int streamID, void * dst, size_t size, size_t * sizeRecvd)
```

 receives data from a remote processor through a communication channel.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See also `rtIOStreamSend` for implementation and performance considerations.

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtIOStreamOpen` | `rtIOStream_wrapper` | `rtIOStreamClose` | `rtIOStreamSend`

rtIOStreamSend

Send data to remote processor

Syntax

```
int rtIOStreamSend(int streamID,const void * src,size_t size,size_t
* sizeSent)
```

Arguments

streamID

A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

src

A pointer to the start of the buffer containing an array of data to transmit

size

Size of data to transmit. For byte-addressable architectures, `size` is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, `size` is measured in number of WORDs, where `sizeof(WORD) == 1`.

sizeSent

Size of data actually transmitted (less than or equal to `size`), or zero if data was not transmitted

Description

```
int rtIOStreamSend(int streamID,const void * src,size_t size,size_t
* sizeSent)
```

sends data to a remote processor through a communication stream.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

Implementation and Performance Considerations

The API for `rtIOStream` functions is designed to be independent of the physical layer across which the data is sent. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for the host-target communication.

For a processor-in-the-loop (PIL) application there is no minimum data rate requirement. However, the higher the data rate, the faster the simulation will run.

In general, a communications device driver will require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel may require specification of which available CAN Node should be used.
- A TCP/IP channel may require a port or static IP address to be configured.
- A CAN channel may require the CAN message ID and priority to be specified.

It is the responsibility of the user who implements the `rtIOStream` driver functions to provide this configuration data, for example by hard-coding it, or by supplying arguments to `rtIOStreamOpen`.

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_rtiostream_script`
- `rtwdemo_custom_pil_script`

See Also

`rtIOStreamOpen` | `rtIOStreamClose` | `rtIOStreamRecv` | `rtiostream_wrapper`

rtIOStreamSend

Send data to remote processor

Syntax

```
int rtIOStreamSend(int streamID,const void * src,size_t size,size_t * sizeSent)
```

Arguments

streamID

A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

src

A pointer to the start of the buffer containing an array of data to transmit.

size

Size of data to transmit. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

sizeSent

Size of data actually transmitted (less than or equal to `size`), or zero if data was not transmitted.

Description

```
int rtIOStreamSend(int streamID,const void * src,size_t size,size_t * sizeSent)
```

 sends data to a remote processor through a communication stream.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

Implementation and Performance Considerations

The API for `rtIOStream` functions is designed to be independent of the physical layer across which the data is sent. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for the host-target communication.

For a processor-in-the-loop (PIL) application there is no minimum data rate requirement. However, the higher the data rate, the faster the simulation will run.

In general, a communications device driver will require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel may require specification of which available CAN Node should be used.
- A TCP/IP channel may require a port or static IP address to be configured.
- A CAN channel may require the CAN message ID and priority to be specified.

It is the responsibility of the user who implements the `rtIOStream` driver functions to provide this configuration data, for example by hard-coding it, or by supplying arguments to `rtIOStreamOpen`.

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtIOStreamOpen` | `rtiostream_wrapper` | `rtIOStreamClose` | `rtIOStreamRecv`

rtiostream_wrapper

Test rtiostream shared library methods

Syntax

```

STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')
STATION_ID = rtiostream_wrapper(SHARED_LIB,'open',p1, v1, p2,
v2, ...)
[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID, DATA,
SIZE)
[RES, DATA_RECVD, SIZE_RECVD] =
rtiostream_wrapper(SHARED_LIB,'recv',ID, SIZE)
RES = rtiostream_wrapper(SHARED_LIB,'close',ID)
rtiostream_wrapper(SHARED_LIB, 'unloadlibrary')

```

Description

`rtiostream_wrapper` enables you to access the methods of an `rtiostream` shared library from MATLAB code, for testing purposes.

`STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')` opens an `rtiostream` communication channel through a shared library, and returns a handle to the channel.

`STATION_ID = rtiostream_wrapper(SHARED_LIB,'open',p1, v1, p2, v2, ...)` opens an `rtiostream` communication channel through a shared library. `p1, v1, ...` are additional parameter value pairs used when opening an `rtiostream` communication channel through a shared library. These arguments are implementation-dependent, that is, they are specific to the shared library being called.

`[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID, DATA, SIZE)` sends `DATA` into the communication channel with handle `ID`, and attempts to send `SIZE` bytes.

`[RES, DATA_RECVD, SIZE_RECVD] = rtiostream_wrapper(SHARED_LIB,'recv',ID, SIZE)` receives up to `SIZE` bytes of `DATA` from the communication channel with handle `ID`.

RES = `rtiostream_wrapper(SHARED_LIB, 'close', ID)` closes the communication channel with handle *ID*.

`rtiostream_wrapper(SHARED_LIB, 'unloadlibrary')` unloads the *SHARED_LIB*, clearing persistent data.

Input Arguments

SHARED_LIB

Name of shared library that implements the required `rtIOStream` functions `rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, and `rtIOStreamClose`. Must be on system path.

Specify shared library:

- *libTCPIP* — For TCP/IP communication. Value of *libTCPIP* depends on your operating system. See `rtwdemo_rtiostream_script`.
- `'libmwrrtiostreamserial.dll'` — For serial communication.

open

Open communication channel

send

Send data into communication channel with handle *ID*

ID

Communication channel handle

DATA

Data to be sent

SIZE

Size of requested data in bytes

recv

Receive data from communication channel with handle *ID*

close

Close communication channel with handle *ID*

unloadlibrary

Unload *SHARED_LIB*

Name-Value Pair Arguments for TCP/IP Communication

p1, v1, ... are optional comma-separated pairs of *Name, Value* arguments, where *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as *Name1, Value1, ..., NameN, ValueN*

The shared library must be *libTCPIP*.

-client

Open *rtiostream* channel as TCP/IP server or client:

- 0 — TCP/IP server
- 1 — TCP/IP client

-port

Specify port number.

-hostname

Specify identifier for host computer, for example, 'localhost'.

-blocking

Specify behavior of call to receive data (call uses input argument *recv*):

- 0 — Polling mode. If data is available, call returns with data. If data is not available, call returns without waiting.
- 1 — Blocking mode. If data is available, call returns with data. If data is not available, call waits for data. Use *recv_timeout_secs* to specify the waiting period.

The default is 0 unless the preprocessor macro *define VXWORKS* exists. In this case, the default is 1.

-recv_timeout_secs

Specify, in seconds, waiting period of call to receive data:

- *X*, an integer greater than zero — Wait for *X* seconds.
- 0 — No waiting period.
- -3 — Wait 10 ms.
- -2 — Wait for default period.
- -1 — Wait indefinitely.

The default for client connections is to wait 1 second. The default for server connections is to wait indefinitely.

Name-Value Pair Arguments for Serial Communication

p1, v1, . . . are optional comma-separated pairs of **Name**, **Value** arguments, where **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as **Name1, Value1, . . . , NameN, ValueN**

The shared library must be 'libmwrTIostreamserial.dll'.

-port

Specify COM port string for serial communication. You must specify bit rate using **-baud**.

-baud

Specify bit rate for serial communication port.

Output Arguments**STATION_ID**

Handle to communication channel. If attempt is unsuccessful, value is -1.

RES

Error flag:

- -1 — Error occurred
- 0 — No error

SIZE_SENT

Number of bytes accepted by communication channel. Can be less than *SIZE*, that is, the requested number of bytes to send.

DATA_RECVD

Data received

SIZE_RECVD

Number of bytes received from channel. Can be less than *SIZE*, that is, the requested number of bytes to send.

Examples

The following examples open communication channels using the supplied TCP/IP and serial communication drivers.

Open `rtiostream` channel `stationA` as a TCP/IP server:

```
stationA = rtiostream_wrapper('libmwrtiostreamtcpip.dll', 'open', ...
    '-client', '0', ...
    '-blocking', '0', ...
    '-port', port_number);
```

Opens `rtiostream` channel `StationB` as a TCP/IP client:

```
stationB = rtiostream_wrapper('libmwrtiostreamtcpip.dll', 'open', ...
    '-client', '1', ...
    '-blocking', '0', ...
    '-port', port_number, ...
    '-hostname', 'localhost');
```

If you use the supplied host-side driver for serial communications (as an alternative to the drivers for TCP/IP), specify the bit rate when you open a channel with a specific port. For example, open channel `stationA` with port `COM1` and bit rate of 9600:

```
stationA = rtiostream_wrapper('libmwrtiostreamserial.dll', 'open', ...
    '-port', 'COM1', ...
    '-baud', '9600');
```

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_rtiostream_script`
- `rtwdemo_custom_pil_script`

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose`

rtiostream_wrapper

Test rtiostream shared library methods

Syntax

```

STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')
STATION_ID = rtiostream_wrapper(SHARED_LIB,'open',p1, v1, p2,
v2, ...)
[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID, DATA,
SIZE)
[RES, DATA_RECVD, SIZE_RECVD] =
rtiostream_wrapper(SHARED_LIB,'recv',ID, SIZE)
RES = rtiostream_wrapper(SHARED_LIB,'close',ID)
rtiostream_wrapper(SHARED_LIB, 'unloadlibrary')

```

Description

`rtiostream_wrapper` enables you to access the methods of an `rtiostream` shared library from MATLAB code, for testing purposes.

`STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')` opens an `rtiostream` communication channel through a shared library, and returns a handle to the channel.

`STATION_ID = rtiostream_wrapper(SHARED_LIB,'open',p1, v1, p2, v2, ...)` opens an `rtiostream` communication channel through a shared library. `p1, v1, ...` are additional parameter value pairs used when opening an `rtiostream` communication channel through a shared library. These arguments are implementation-dependent, that is, they are specific to the shared library being called.

`[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID, DATA, SIZE)` sends `DATA` into the communication channel with handle `ID`, and attempts to send `SIZE` bytes.

`[RES, DATA_RECVD, SIZE_RECVD] = rtiostream_wrapper(SHARED_LIB,'recv',ID, SIZE)` receives up to `SIZE` bytes of `DATA` from the communication channel with handle `ID`.

RES = `rtiostream_wrapper(SHARED_LIB, 'close', ID)` closes the communication channel with handle *ID*.

`rtiostream_wrapper(SHARED_LIB, 'unloadlibrary')` unloads the *SHARED_LIB*, clearing persistent data.

Input Arguments

SHARED_LIB

Name of shared library that implements the required `rtIOStream` functions `rtIOStreamOpen`, `rtIOStreamSend`, `rtIOStreamRecv`, and `rtIOStreamClose`. Must be on system path.

Specify shared library:

- *libTCPIP* — For TCP/IP communication. Value of *libTCPIP* depends on your operating system. See `rtwdemo_rtiostream_script`.
- `'libmwrrtiostreamserial.dll'` — For serial communication.

open

Open communication channel

send

Send data into communication channel with handle *ID*

ID

Communication channel handle

DATA

Data to be sent

SIZE

Size of requested data in bytes

recv

Receive data from communication channel with handle *ID*

close

Close communication channel with handle *ID*

unloadlibrary

Unload *SHARED_LIB*

Name-Value Pair Arguments for TCP/IP Communication

p1, v1, ... are optional comma-separated pairs of *Name, Value* arguments, where *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as *Name1, Value1, ..., NameN, ValueN*

The shared library must be *libTCPIP*.

-client

Open *rtiostream* channel as TCP/IP server or client:

- 0 — TCP/IP server
- 1 — TCP/IP client

-port

Specify port number.

-hostname

Specify identifier for host computer, for example, 'localhost'.

-blocking

Specify behavior of call to receive data (call uses input argument *recv*):

- 0 — Polling mode. If data is available, call returns with data. If data is not available, call returns without waiting.
- 1 — Blocking mode. If data is available, call returns with data. If data is not available, call waits for data. Use *recv_timeout_secs* to specify the waiting period.

The default is 0 unless the preprocessor macro *define VXWORKS* exists. In this case, the default is 1.

-recv_timeout_secs

Specify, in seconds, waiting period of call to receive data:

- *X*, an integer greater than zero — Wait for *X* seconds.
- 0 — No waiting period.
- -3 — Wait 10 ms.
- -2 — Wait for default period.
- -1 — Wait indefinitely.

The default for client connections is to wait 1 second. The default for server connections is to wait indefinitely.

Name-Value Pair Arguments for Serial Communication

p1, v1, . . . are optional comma-separated pairs of **Name**, **Value** arguments, where **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as **Name1, Value1, . . . , NameN, ValueN**

The shared library must be 'libmwrTIostreamserial.dll'.

-port

Specify COM port string for serial communication. You must specify bit rate using **-baud**.

-baud

Specify bit rate for serial communication port.

Output Arguments**STATION_ID**

Handle to communication channel. If attempt is unsuccessful, value is -1.

RES

Error flag:

- -1 — Error occurred
- 0 — No error

SIZE_SENT

Number of bytes accepted by communication channel. Can be less than *SIZE*, that is, the requested number of bytes to send.

DATA_RECVD

Data received

SIZE_RECVD

Number of bytes received from channel. Can be less than *SIZE*, that is, the requested number of bytes to send.

Examples

The following examples open communication channels using the supplied TCP/IP and serial communication drivers.

Open `rtiostream` channel `stationA` as a TCP/IP server:

```
stationA = rtiostream_wrapper('libmwrtiostreamtcpip.dll', 'open', ...
                             '-client', '0', ...
                             '-blocking', '0', ...
                             '-port', port_number);
```

Opens `rtiostream` channel `StationB` as a TCP/IP client:

```
stationB = rtiostream_wrapper('libmwrtiostreamtcpip.dll', 'open', ...
                              '-client', '1', ...
                              '-blocking', '0', ...
                              '-port', port_number, ...
                              '-hostname', 'localhost');
```

If you use the supplied host-side driver for serial communications (as an alternative to the drivers for TCP/IP), specify the bit rate when you open a channel with a specific port. For example, open channel `stationA` with port `COM1` and bit rate of 9600:

```
stationA = rtiostream_wrapper('libmwrtiostreamserial.dll', 'open', ...
                              '-port', 'COM1', ...
                              '-baud', '9600');
```

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamClose` | `rtIOStreamRecv`

rtw.codegenObjectives.Objective class

Package: rtw.codegenObjectives

Customize code generation objectives

Description

An `rtw.codegenObjectives.Objective` object creates a code generation objective.

Construction

`rtw.codegenObjectives.Objective`

Create custom code generation objectives

Methods

`addCheck`

Add checks

`addParam`

Add parameters

`excludeCheck`

Exclude checks

`modifyInheritedParam`

Modify inherited parameter values

`register`

Register objective

`removeInheritedCheck`

Remove inherited checks

`removeInheritedParam`

Remove inherited parameters

setObjectiveName

Specify objective name

Copy Semantics

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

Examples

Create a custom objective named **Reduce RAM Example**. The following code is the contents of the `sl_customization.m` file that you create.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end

function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'mathworks.design.UnconnectedLinesPorts');
addCheck(obj, 'mathworks.design.Update');

%Register the objective
register(obj);

end
```


See Also

How To

- “Create Custom Objectives”

rtw.codegenObjectives.Objective

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Create custom code generation objectives

Syntax

```
obj = rtw.codegenObjectives.Objective('objID')  
obj = rtw.codegenObjectives.Objective('objID', 'base_objID')
```

Description

obj = rtw.codegenObjectives.Objective('objID') creates an objective object, *obj*.

obj = rtw.codegenObjectives.Objective('objID', 'base_objID') creates an object, *obj*, for a new objective that is identical to an existing objective. You can then modify the new objective to meet your requirements.

Input Arguments

<i>objID</i>	A permanent, unique identifier for the objective. <ul style="list-style-type: none">You must have<ul style="list-style-type: none"><i>objID</i>.The value of <i>objID</i> must remain constant.When you refresh your customizations, if <i>objID</i> is not unique, Simulink generates an error.
<i>base_objID</i>	The identifier of the objective that you want to base the new objective on.

Examples

Create a new objective:

```
obj = rtw.codegenObjectives.Objective('ex_ram_1');
```

Create a new objective based on the existing `Execution efficiency` objective:

```
obj = rtw.codegenObjectives.Objective('ex_my_efficiency_1', 'Execution efficiency');
```

How To

- “Create Custom Objectives”

RTW.configSubsystemBuild

Package: RTW

Configure C function prototype or C++ class interface for right-click build of specified subsystem

Syntax

RTW.configSubsystemBuild(*block*)

Description

RTW.configSubsystemBuild(*block*) opens a graphical user interface where you can configure either C function prototype information or C++ class interface information for right-click builds of a specified nonvirtual subsystem. A dialog box opens based on the **Language** and **Code interface packaging** values selected for your model on the **Code Generation** and **Code Generation > Interface** panes of the Configuration Parameters dialog box.

To configure and generate C++ class interfaces for a nonvirtual subsystem, you must

- Select the system target file `ert.tlc` for the model.
- Select the **Language** parameter value **C++** for the model.
- Select the **Code interface packaging** parameter value **C++ class** for the model.
- Make sure that the subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

Input Arguments

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

More About

- “Configure Function Prototypes for Nonvirtual Subsystems”
- “Function Prototype Control”
- “Configure C++ Class Interfaces for Nonvirtual Subsystems”
- “C++ Class Interface Control”

rtw.connectivity.ComponentArgs

Provide parameters to each target connectivity component

Syntax

```
componentArgs = rtw.connectivity.ComponentArgs (componentPath,  
componentCodePath, componentCodeName, applicationCodePath)
```

Description

`componentArgs = rtw.connectivity.ComponentArgs (componentPath, componentCodePath, componentCodeName, applicationCodePath)` returns a handle to an object that provides methods for getting information about the source component (e.g., the MATLAB function under test) and the target application (e.g., the PIL application).

For methods, see the following table.

Method	Syntax and Description
<code>getComponentPath</code>	<code>componentPath = obj.getComponentPath</code> Returns the Simulink system path of the source component (e.g., the path of the referenced model that is under test).
<code>getComponentCodePath</code>	<code>componentCodePath = obj.getComponentCodePath</code> Returns the Embedded Coder code generation directory path associated with the source component (e.g., the code generation directory of the referenced model that is under test).
<code>getComponentCodeName</code>	<code>componentCodeName = obj.getComponentCodeName</code>

Method	Syntax and Description
	Returns the component name used for code generation.
getApplicationCodePath	<pre>applicationCodePath = obj.getApplicationCodePath</pre>
	Returns the directory path associated with the target application (e.g., the path associated with the PIL application).

See `rtw.connectivity.Config` for more information.

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtw.connectivity.Config`

rtw.connectivity.ComponentArgs

Provide parameters to each target connectivity component

Syntax

```
componentArgs = rtw.connectivity.ComponentArgs(componentPath,  
componentCodePath, componentCodeName, applicationCodePath)
```

Description

`componentArgs = rtw.connectivity.ComponentArgs(componentPath, componentCodePath, componentCodeName, applicationCodePath)` returns a handle to an object that provides methods for getting information about the source component (e.g., the MATLAB function under test) and the target application (e.g., the PIL application).

Method	Syntax and Description
getComponentPath	<code>componentPath = obj.getComponentPath</code>
	Returns the system path of the source component (e.g., the path of the function that is under test).
getComponentCodePath	<code>componentCodePath = obj.getComponentCodePath</code>
	Returns the code generation folder path associated with the source component (e.g., the code generation folder of the MATLAB function that is under test).
getComponentCodeName	<code>componentCodeName = obj.getComponentCodeName</code>
	Returns the component name used for code generation.

Method	Syntax and Description
getApplicationCodePath	applicationCodePath = obj.getApplicationCodePath Returns the folder path associated with the target application (e.g., the path associated with the PIL application).

More About

- “Create PIL Target Connectivity Configuration”

See Also

rtw.connectivity.Config

rtw.connectivity.Config

Define connectivity implementation, comprising builder, launcher, and communicator components

Syntax

```
rtw.connectivity.Config(componentArgs, builder, launcher,  
communicator)
```

Description

Constructor	Description
Config	Wrapper for the connectivity component classes builder, launcher and communicator.

Constructor Arguments	
componentArgs	rtw.connectivity.ComponentArgs object.
builder	rtw.connectivity.Builder, for example, rtw.connectivity.MakefileBuilder object.
launcher	rtw.connectivity.Launcher object.
communicator	rtw.connectivity.Communicator, for example, rtw.connectivity.RtIOStreamHostCommunicator object.

rtw.connectivity.Config(componentArgs, builder, launcher, communicator) creates an rtw.connectivity.Config object.

To define a connectivity implementation:

- 1 You must create a subclass of `rtw.connectivity.Config` that creates instances of your connectivity component classes:

- `rtw.connectivity.MakefileBuilder`
- `rtw.connectivity.Launcher`
- `rtw.connectivity.RtIOStreamHostCommunicator`

You can see an example `ConnectivityConfig.m`, used in `rtwdemo_custom_pil_script`.

- 2 Define the constructor for your subclass as follows:

```
function this = MyConfig(componentArgs)
```

When Simulink creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you may want to create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

- 3 After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration. For example:

```
% Call superclass constructor to register components
this@rtw.connectivity.Config(componentArgs,...
builder, launcher, communicator);
```

- 4 Optionally, for execution time profiling, use the `setTimer` method to register your hardware timer. For example, if you specified the “timer in a code replacement table”, insert the following line:

```
this.setTimer('MyCr1Table')
```

`MyCr1Table` is the name of the code replacement table, which must be in a location on the MATLAB search path.

Register your subclass name, for example, `MyPIL.ConnectivityConfig` to Simulink by using the class `rtw.connectivity.ConfigRegistry`. This uses the `sl_customization.m` mechanism to register your connectivity configuration.

The PIL infrastructure instantiates your subclass as required. The `sl_customization.m` mechanism helps in specifying a suitable connectivity

configuration for use with a particular PIL component (and its configuration set). The subclass can also perform additional validation on construction. For example, you can use the `componentPath` returned by the `getComponentPath` method of the `componentArgs` constructor argument to query and validate parameters associated with the PIL component under test.

For supported hardware implementation settings and other support information, see “SIL and PIL Simulation Support and Limitations”.

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_custom_pil_script`
- “Specify Hardware Timer”

See Also

`rtw.connectivity.MakefileBuilder` | `rtw.connectivity.Launcher`
| `rtw.connectivity.RtIOStreamHostCommunicator` |
`rtw.connectivity.ComponentArgs`

rtw.connectivity.Config

Define connectivity implementation, comprising builder, launcher, and communicator components

Syntax

```
rtw.connectivity.Config(componentArgs, builder, launcher,
communicator)
```

Description

Constructor	Description
Config	Wrapper for the connectivity component classes builder, launcher and communicator.

Constructor Arguments	
componentArgs	rtw.connectivity.ComponentArgs object.
builder	rtw.connectivity.Builder, for example, rtw.connectivity.MakefileBuilder object.
launcher	rtw.connectivity.Launcher object.
communicator	rtw.connectivity.Communicator, for example, rtw.connectivity.RtIOStreamHostCommunicator object.

rtw.connectivity.Config(componentArgs, builder, launcher, communicator) defines a connectivity implementation:

- 1 You must create a subclass of rtw.connectivity.Config that creates instances of your connectivity component classes:

- `rtw.connectivity.MakefileBuilder`
- `rtw.connectivity.Launcher`
- `rtw.connectivity.RtIOStreamHostCommunicator`

2 Define the constructor for your subclass as follows:

```
function this = MyConfig(componentArgs)
```

When the software creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you may want to create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

3 After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration. For example:

```
% Call superclass constructor to register components  
this@rtw.connectivity.Config(componentArgs,...  
builder, launcher, communicator);
```

4 Optionally, for execution time profiling, use the `setTimer` method to register your hardware timer. For example, if you specified the “timer in a code replacement table”, insert the following line:

```
this.setTimer(MyCr1Table)
```

MyCr1Table is the name of the code replacement table, which must be in a location on the MATLAB search path.

Register your subclass name (e.g., `MyPIL.ConnectivityConfig`) with MATLAB using the `rtw.connectivity.ConfigRegistry` class. You require an `rtwTargetInfo.m` file to register your connectivity configuration.

The PIL infrastructure instantiates your subclass as required. The file `rtwTargetInfo.m` specifies a suitable connectivity configuration for use with a particular PIL component (and its configuration set). The subclass can also perform additional validation on construction. For example, you can use the `componentPath` returned by the `getComponentPath` method of the `componentArgs` constructor argument to query and validate parameters associated with the PIL code under test.

More About

- “Create PIL Target Connectivity Configuration”
- “Specify Hardware Timer”

See Also

`rtw.connectivity.Launcher` | `rtw.connectivity.ComponentArgs` |
`rtw.connectivity.MakefileBuilder` |
`rtw.connectivity.RtIOStreamHostCommunicator`

rtw.connectivity.ConfigRegistry

Register connectivity configuration

Syntax

```
config = rtw.connectivity.ConfigRegistry
```

Description

Use this class to register your connectivity configuration with Simulink by using the `sl_customization.m` mechanism. The connectivity configuration is registered by a call to `registerTargetInfo` inside a `sl_customization.m` file.

Create or add to your `sl_customization.m` file as shown in the “Examples” on page 1-422 section, and place the file on the MATLAB path. Simulink software reads the `sl_customization.m` when it starts, and registers your connectivity configuration. This step also defines the set of Simulink models that the new connectivity configuration is compatible with.

A connectivity configuration must have a unique name and be associated with a connectivity implementation class (a subclass of `rtw.connectivity.Config`). The properties of the configuration (for example, `SystemTargetFile`) define the set of Simulink models that the connectivity implementation class is compatible with. The properties are shown in the following table.

Properties of `rtw.connectivity.ConfigRegistry`

Property Name	Description
ConfigName	Unique string name for this configuration
ConfigClass	Full class name of the connectivity implementation (e.g. <code>rtw.pil.myConnectivityConfig</code>) to register.
SystemTargetFile	Cell array of strings listing System Target Files that support this <code>ConfigRegistry</code> .

Property Name	Description
	<p>An empty cell array matches any System Target File.</p> <p>The model's <code>SystemTargetFile</code> configuration parameter is validated against this cell array to determine if this <code>ConfigRegistry</code> is valid for use.</p>
TemplateMakefile	<p>Cell array of strings listing Template Makefiles that support this <code>ConfigRegistry</code>. An empty cell array matches any Template Makefile and nonmakefile based targets (<code>GenerateMakefile: off</code>).</p> <p>The model's <code>TemplateMakefile</code> configuration parameter is validated against this cell array to determine if this <code>ConfigRegistry</code> is valid for use.</p> <hr/> <p>Note: If you use a toolchain to build the generated code, do not specify the <code>TemplateMakefile</code> configuration parameter. Instead, specify the <code>Toolchain</code> configuration parameter.</p>
Toolchain	<p>Cell array of strings listing toolchains that support this <code>ConfigRegistry</code>. An empty cell array matches any toolchain. The model's <code>Toolchain</code> configuration parameter is validated against this cell array to determine if this <code>ConfigRegistry</code> is valid for use.</p> <hr/> <p>Note: If you do not use a toolchain to build the generated code, do not specify the <code>Toolchain</code> configuration parameter. Instead, specify the <code>TemplateMakefile</code> configuration parameter.</p>

Property Name	Description
TargetHWDeviceType	Cell array of strings listing Hardware Device Types that support this ConfigRegistry. An empty cell array matches any Hardware Device Type. The model's TargetHWDeviceType configuration parameter is validated against this cell array to determine if this ConfigRegistry is valid for use.

Examples

The following code shows an example `sl_customization.m` registration. You must use the `sl_customization.m` file structure shown in the example following. You must call the `registerTargetInfo` function exactly as shown.

```
function sl_customization(cm)
% SL_CUSTOMIZATION for PIL connectivity config:...
% mypil.ConnectivityConfig

% Copyright 2008 The MathWorks, Inc.

cm.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

config = rtw.connectivity.ConfigRegistry;
config.ConfigName = 'My PIL Example';
config.ConfigClass = 'mypil.ConnectivityConfig';

% match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};

% If you use a toolchain to build your generated code
% you must specify the config.Toolchain property to match
% your Simulink model toolchain setting
% Otherwise, for a non-toolchain approach, match the TMF
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
```

```

        'ert_vc.tmf', ...
        'ert_vcx64.tmf', ...
        'ert_lcc.tmf'};

% match regular 32-bit machines and Custom for e.g. ...
% 64-bit Linux
config.TargetHWDeviceType = {'Generic->32-bit x86 ...
                             compatible'
                             'Generic->Custom'};

```

You must configure the file to perform the following steps when Simulink software starts:

- 1 Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example,

```
config = rtw.connectivity.ConfigRegistry;
```

- 2 Assign a connectivity configuration name to the `ConfigName` property of the object. For example,

```
config.ConfigName = 'My PIL Example';
```

- 3 Associate the connectivity configuration with the connectivity API implementation (created in step 1). For example,

```
config.ConfigClass = 'mypil.ConnectivityConfig';
```

- 4 Define compatible models for this target connectivity configuration, by setting the `SystemTargetFile`, `TemplateMakefile` (non-toolchain approach) and `TargetHWDeviceType` properties of the object. For example,

```
% match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};
```

```
% match the TMF
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vc.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};
```

```
% match regular 32-bit machines and Custom for e.g. ...
% 64-bit Linux
config.TargetHWDeviceType = {'Generic->32-bit x86 ...
                             compatible'
                             'Generic->Custom'};
```

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_custom_pil_script`

See Also

`rtw.connectivity.Config`

rtw.connectivity.ConfigRegistry

Register target connectivity configuration

Syntax

```
config = rtw.connectivity.ConfigRegistry
```

Description

`config = rtw.connectivity.ConfigRegistry` returns a handle to an object that is required for registering your target connectivity configuration with MATLAB.

Use an `rtwTargetInfo.m` file to create this class. In the `rtwTargetInfo.m` file, the connectivity configuration is registered by a call to `registerTargetInfo`. You must place the `rtwTargetInfo.m` file on the MATLAB search path.

A target connectivity configuration must have a unique name and be associated with a connectivity implementation class (a subclass of `rtw.connectivity.Config`). The properties of the configuration (for example, `TargetHWDeviceType`) define the MATLAB code that is compatible with the connectivity implementation class.

Properties of `rtw.connectivity.ConfigRegistry`

Property Name	Description
<code>ConfigName</code>	Unique string name for this configuration
<code>ConfigClass</code>	Full class name of the connectivity implementation e.g. <code>rtw.pil.myConnectivityConfig</code> .
<code>Toolchain</code>	Cell array of strings listing toolchains that support this <code>ConfigRegistry</code> . An empty cell array matches any toolchain. The MATLAB Coder <code>Toolchain</code> configuration setting is validated against this cell array to determine if this <code>ConfigRegistry</code> is valid for use.

Property Name	Description
TargetHWDeviceType	<p>Cell array of strings listing hardware device types that support this <code>ConfigRegistry</code>. An empty cell array matches any hardware device type.</p> <p>The MATLAB Coder <code>TargetHWDeviceType</code> configuration setting is validated against this cell array to determine if this <code>ConfigRegistry</code> is valid for use.</p>

MATLAB Coder configuration settings are specified through the project interface or the `coder.EmbeddedCodeConfig` object that `codegen` uses.

Examples

The following code shows an example `rtwTargetInfo.m` file. You must call the `registerTargetInfo` function exactly as shown.

```
function rtwTargetInfo(tr)
% Register PIL connectivity config: mypil.ConnectivityConfig

tr.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

% Create object for connectivity configuration
config = rtw.connectivity.ConfigRegistry;
% Assign connectivity configuration name
config.ConfigName = 'My PIL Example';
% Associate the connectivity configuration with the connectivity
% API implementation
config.ConfigClass = 'mypil.ConnectivityConfig';

% Specify toolchains for host-based PIL
config.Toolchain = rtw.connectivity.Utils.getHostToolchainNames;

% Through the TargetHWDeviceType property, define compatible code
```

```
% for the target connectivity configuration
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'ARM Compatible->ARM Cortex'};
```

When MATLAB software runs, the file performs the following steps :

- 1 Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example:

```
config = rtw.connectivity.ConfigRegistry;
```

- 2 Assign a connectivity configuration name to the `ConfigName` property of the object. For example:

```
config.ConfigName = 'My PIL Example';
```

- 3 Associate the connectivity configuration with the connectivity API implementation created in step 1. For example:

```
config.ConfigClass = 'mypil.ConnectivityConfig';
```

- 4 Define compatible code for this target connectivity configuration, by setting the `TargetHWDeviceType` property of the object. For example:

```
config.TargetHWDeviceType = {'Generic->32-bit x86 compatible' ...
                             'Generic->Custom' ...
                             'ARM Compatible->ARM Cortex'};
```

More About

- “Create PIL Target Connectivity Configuration”
- “Register Code Replacement Mappings”

See Also

`rtw.connectivity.Config`

rtw.connectivity.Launcher

Control downloading, starting and resetting of a target application

Syntax

```
rtw.connectivity.Launcher(componentArgs)
```

Description

Constructor	Description
Launcher	Controls execution of an application on target hardware.

`rtw.connectivity.Launcher(componentArgs)` controls the download, start and reset of an application, for example, a PIL application.

You can also use `rtw.connectivity.Launcher(componentArgs, builder)`, which provides the Launcher access to a Builder object through the `getBuilder` method. However, support for this approach will cease in a future release.

You must make a subclass and implement the `startApplication` and `stopApplication` methods.

You can implement a destructor method that cleans up resources (for example, a handle to a third-party download tool) when this object is cleared from memory. There is significant flexibility in how the `startApplication` and `stopApplication` methods can be implemented.

For methods, see the following table.

Method	Syntax and Description
<code>getComponentArgs</code>	<code>componentArgs = obj.getComponentArgs</code> Returns the <code>rtw.connectivity.ComponentArgs</code> object associated with the Launcher object.
<code>setExe</code>	<code>setExe(exe)</code>

Method	Syntax and Description
	Specify the application to run on the target
getExe	<pre>exe=getExe()</pre> <p>Returns the application running on the target</p>
startApplication	<pre>obj.startApplication</pre> <p>Abstract method that you must implement in a subclass.</p> <p>Called by Simulink to start execution of the target application.</p> <p>Simulink calls the <code>setExe</code> method, which specifies the target application to launch. To obtain this application, use the <code>getExe</code> method. For example:</p> <pre>exe = getExe()</pre> <p>The <code>startApplication</code> method must reset the application to its initial state by ensuring that external and static (global) variables are zero initialized.</p>
stopApplication	<pre>obj.stopApplication</pre> <p>Abstract method that you must implement in a subclass.</p> <p>Called by Simulink to stop execution of the target application.</p>
getBuilder	<pre>builder = obj.getBuilder</pre> <p>Returns the <code>rtw.connectivity.Builder</code> object associated with the Launcher object.</p>

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_custom_pil_script`

rtw.connectivity.Launcher

Control downloading, starting and resetting of a target application

Syntax

```
rtw.connectivity.Launcher(componentArgs)
```

Description

Constructor	Description
Launcher	Controls execution of an application on target hardware.

`rtw.connectivity.Launcher(componentArgs)` controls the download, start and reset of an application, for example, a PIL application.

You must make a subclass and implement the `startApplication` and `stopApplication` methods.

You can implement a destructor method that cleans up resources (for example, a handle to a third-party download tool) when this object is cleared from memory. There is flexibility in how the `startApplication` and `stopApplication` methods can be implemented.

For methods, see the following table.

Method	Syntax and Description
getComponentArgs	<code>componentArgs = obj.getComponentArgs</code>
	Returns the <code>rtw.connectivity.ComponentArgs</code> object associated with the <code>Launcher</code> object.
setExe	<code>setExe(exe)</code>
	Specify the application to run on the target
getExe	<code>exe=getExe()</code>
	Returns the application running on the target

Method	Syntax and Description
startApplication	obj.startApplication
	<p>Abstract method that you must implement in a subclass.</p> <p>Called by MATLAB to start execution of the target application.</p> <p>MATLAB calls the <code>setExe</code> method, which specifies the target application to launch. To obtain this application, use the <code>getExe</code> method. For example:</p> <pre>exe = getExe()</pre> <p>The <code>startApplication</code> method must reset the application to its initial state by ensuring that external and static (global) variables are zero initialized.</p>
stopApplication	obj.stopApplication
	<p>Abstract method that you must implement in a subclass.</p> <p>Called by MATLAB to stop execution of the target application.</p>

More About

- “Create PIL Target Connectivity Configuration”

See Also

rtw.connectivity.Config | rtw.connectivity.MakefileBuilder

rtw.connectivity.MakefileBuilder

Configure makefile-based build process

Syntax

```
rtw.connectivity.MakefileBuilder(componentArgs,  
targetApplicationFramework, exeExtension)
```

Description

Constructor	Description
MakefileBuilder	Control makefile-based build process.

Constructor Arguments	
componentArgs	rtw.connectivity.ComponentArgs
TargetApplicationFramework	rtw.pil.RtIOStreamApplicationFramework (e.g. MyPIL.TargetFramework)
exeExtension	Filename extension of an executable for the target system. The extension depends on the makefile and compiler that are called by the MakefileBuilder . These are defined by the template makefile specified by the source component (e.g., the referenced model under test). For an embedded target the extension may be <code>' .elf '</code> , <code>' .abs '</code> , <code>' .sre '</code> , <code>' .hex '</code> , or others. For a Windows host-based target the extension is <code>' .exe '</code> . For a UNIX [®] host-based target the extension is empty, <code>' '</code> .

`rtw.connectivity.MakefileBuilder(componentArgs, targetApplicationFramework, exeExtension)` returns a handle to an object that controls the toolchain-based build process.

MakefileBuilder controls the customizable makefile-based build process supporting the creation of custom applications (e.g. a PIL application) that interface with a Simulink component such as a referenced model (represented as a collection of binary libraries).

To build the PIL application, you must provide a template makefile that includes the target `MAKEFILEBUILDER_TGT`. You can use the standard TMF files, e.g., `ert_unix.tmf` or `ert_vc.tmf`.

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_custom_pil_script`

See Also

`rtw.pil.RtIOStreamApplicationFramework` |
`rtw.connectivity.ComponentArgs`

rtw.connectivity.MakefileBuilder

Configure toolchain-based build process

Syntax

```
builder = rtw.connectivity.MakefileBuilder(componentArgs,  
targetApplicationFramework, exeExtension)
```

Description

Constructor	Description
MakefileBuilder	Control toolchain-based build process.

Constructor Arguments	
componentArgs	An <code>rtw.connectivity.ComponentArgs</code> object
TargetApplicationFramework	An <code>rtw.pil.RtIOStreamApplicationFramework</code> object
exeExtension	Name extension of executable file for target system. The extension depends on the toolchain defined by <code>rtw.connectivity.ConfigRegistry</code> . For an embedded target, the extension can be, for example, <code>'.elf'</code> , <code>'.abs'</code> , <code>'.sre'</code> , or <code>'.hex'</code> . For a Windows host-based target the extension is <code>'.exe'</code> . For a UNIX host-based target the extension is empty, <code>''</code> .

`builder = rtw.connectivity.MakefileBuilder(componentArgs, targetApplicationFramework, exeExtension)` returns a handle to an object that controls the toolchain-based build process.

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtw.connectivity.Config` | `rtw.connectivity.Launcher` |
`rtw.connectivity.ComponentArgs` | `rtw.connectivity.ConfigRegistry` |
`rtw.pil.RtIOStreamApplicationFramework`

rtw.connectivity.RtIOStreamHostCommunicator

Configure host-side communications

Syntax

```
rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher, rtiostreamLib)
```

Description

Constructor	Description
RtIOStreamHostCommunicator	Configure host-side communications with the target by loading and initializing a shared library that implements the <code>rtiostream</code> functions.

Constructor Arguments	
<code>componentArgs</code>	A <code>rtw.connectivity.ComponentArgs</code> object.
<code>launcher</code>	A <code>rtw.connectivity.Launcher</code> object.
<code>rtiostreamLib</code>	An <code>rtiostream</code> shared library that implements the host side of host-target communications.

`rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher, rtiostreamLib)` returns a handle to an object for configuring host-side communications with the target. The object loads and initializes a shared library that implements the `rtiostreamfunctions`.

Embedded Coder provides an implementation of this shared library to support TCP/IP communications between host and target, as well as a version for serial communications. With TCP/IP or serial, you need only supply the target-side drivers.

For other communications protocols (e.g. USB), you must supply a shared library for the host-side of the communications link as well as the target-side drivers.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have two options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments to supply to the `rtiostream` shared library.
- Alternatively, create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. Consider this when more complex configuration is required. For example, the subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the TCP/IP port number on which the executable application is serving, or you could use a subclass to specify a serial port number, or specify verbose or silent operation.

Methods	
<code>setTimeoutRecvSecs</code>	Sets the timeout value for reading data.
<code>hostCommunicator.setTimeoutRecvSecs(<i>timeout</i>)</code> configures data reading to time out if no new data is received for a period of greater than <code>timeout</code> seconds.	
<code>setInitCommsTimeout</code>	Sets the timeout value for initial setup of the communications channel.
<code>hostCommunicator.setInitCommsTimeout(<i>timeout</i>)</code> For some targets you may need to set a timeout value for initial setup of the communications channel. For example, the target processor may take a few seconds before it is ready to open its side of the communications channel. If you set a nonzero timeout value then the communicator repeatedly tries to open the communications channel until the timeout value is reached.	

More About

- “Create PIL Target Connectivity Configuration”
- `rtwdemo_custom_pil_script`

See Also

`rtw.connectivity.ComponentArgs` | `rtw.connectivity.Launcher` | `rtiostream_wrapper`

rtw.connectivity.RtIOStreamHostCommunicator

Configure host-side communications

Syntax

```
rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher,  
rtiostreamLib)
```

Description

Constructor	Description
RtIOStreamHostCommunicator	Configure host-side communications with the target by loading and initializing a shared library that implements the <code>rtiostream</code> functions.

Constructor Arguments	
<code>componentArgs</code>	A <code>rtw.connectivity.ComponentArgs</code> object.
<code>launcher</code>	A <code>rtw.connectivity.Launcher</code> object.
<code>rtiostreamLib</code>	An <code>rtiostream</code> shared library that implements the host side of host-target communications.

`rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, launcher, rtiostreamLib)` returns a handle to an object for configuring host-side communications with the target. The object loads and initializes a shared library that implements the `rtiostream` functions.

Embedded Coder provides an implementation of this shared library to support TCP/IP communications between host and target, as well as a version for serial communications. With TCP/IP or serial, you need only supply the target-side drivers.

For other communications protocols (e.g. USB), you must supply a shared library for the host-side of the communications link as well as the target-side drivers.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have two options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments to supply to the `rtiostream` shared library.
- Create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. Use this option when more complex configuration is required. For example, the subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the TCP/IP port number on which the executable application is serving, or you could use a subclass to specify a serial port number, or specify verbose or silent operation.

Methods	
<code>setTimeoutRecvSecs</code>	Sets the timeout value for reading data.
<code>hostCommunicator.setTimeoutRecvSecs(<i>timeout</i>)</code> configures data reading to time out if no new data is received for a period of greater than <i>timeout</i> seconds.	
<code>setInitCommsTimeout</code>	Sets the timeout value for initial setup of the communications channel.
<code>hostCommunicator.setInitCommsTimeout(<i>timeout</i>)</code> For some targets you might need to set a timeout value for initial setup of the communications channel. For example, the target processor may take a few seconds before it is ready to open its side of the communications channel. If you set a nonzero timeout value then the communicator repeatedly tries to open the communications channel until the timeout value is reached.	

More About

- “Create PIL Target Connectivity Configuration”

See Also

`rtw.connectivity.Launcher` | `rtw.connectivity.ComponentArgs` | `rtiostream_wrapper`

RTW.getClassInterfaceSpecification

Package: RTW

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

Syntax

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

Description

`obj = RTW.getClassInterfaceSpecification(modelName)` returns a handle to a model-specific C++ class interface control object.

Input Arguments

<i>modelName</i>	String specifying the name of a loaded ERT-based Simulink model.
------------------	--

Output Arguments

obj	Handle to the C++ class interface control object associated with the specified model. If the model does not have an associated C++ class interface control object, the function returns [].
-----	--

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your

C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

More About

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

RTW.getFunctionSpecification

Package: RTW

Get handle to model-specific C prototype function control object

Syntax

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

Description

obj = RTW.getFunctionSpecification(*modelName*) returns a handle to the model-specific C function prototype control object.

Input Arguments

<i>modelName</i>	String specifying the name of a loaded ERT-based Simulink model.
------------------	--

Output Arguments

<i>obj</i>	Handle to the model-specific C prototype function control object associated with the specified model. If the model does not have an associated function control object, the function returns [].
------------	---

Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. Once you validate and apply your changes, you can generate code based on your

C function prototype modifications. See “Configure Function Prototypes Using Graphical Interfaces” in the Embedded Coder documentation.

More About

- “Function Prototype Control”

RTW.ModelCPPArgsClass class

Package: RTW

Superclasses: RTW.ModelCPPClass

Control C++ class interfaces for models using I/O arguments style step method

Description

The `ModelCPPArgsClass` class provides objects that describe C++ class interfaces for models using an I/O arguments style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

Construction

`RTW.ModelCPPArgsClass`

Create C++ class interface object for configuring model class with I/O arguments style step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following methods.

`getArgCategory`

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

`getArgName`

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

`getArgPosition`

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

<code>getArgQualifier</code>	Get argument type qualifier for Simulink model port from model-specific C++ class interface
<code>runValidation</code>	Validate model-specific C++ class interface against Simulink model
<code>setArgCategory</code>	Set argument category for Simulink model port in model-specific C++ class interface
<code>setArgName</code>	Set argument name for Simulink model port in model-specific C++ class interface
<code>setArgPosition</code>	Set argument position for Simulink model port in model-specific C++ class interface
<code>setArgQualifier</code>	Set argument type qualifier for Simulink model port in model-specific C++ class interface

Copy Semantics

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

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

RTW.ModelCPPArgsClass

Class: RTW.ModelCPPArgsClass

Package: RTW

Create C++ class interface object for configuring model class with I/O arguments style step method

Syntax

```
obj = RTW.ModelCPPArgsClass
```

Description

obj = RTW.ModelCPPArgsClass returns a handle, *obj*, to a newly created object of class RTW.ModelCPPArgsClass.

Output Arguments

<i>obj</i>	Handle to a newly created C++ class interface object for configuring a model class with an I/O arguments style step method. The object has not yet been configured or attached to an ERT-based Simulink model.
------------	--

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”

- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

RTW.ModelCPPClass class

Package: RTW

Control C++ class interfaces for models

Description

The `ModelCPPClass` class is the base class for the classes `ModelCPPArgsClass` and `ModelCPPVoidClass`, which provide objects that describe C++ class interfaces for models using either an I/O arguments style step method or a `void-void` style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

Construction

To access the methods of this class, use the constructor for either `RTW.ModelCPPArgsClass` or `RTW.ModelCPPVoidClass`.

Methods

<code>attachToModel</code>	Attach model-specific C++ class interface to loaded ERT-based Simulink model
<code>getClassName</code>	Get class name from model-specific C++ class interface
<code>getDefaultConf</code>	Get default configuration information for model-specific C++ class interface from Simulink model
<code>getNamespace</code>	Get namespace from model-specific C++ class interface

getNumArgs	Get number of step method arguments from model-specific C++ class interface
getStepMethodName	Get step method name from model-specific C++ class interface
setClassName	Set class name in model-specific C++ class interface
setNamespace	Set namespace in model-specific C++ class interface
setStepMethodName	Set step method name in model-specific C++ class interface

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

RTW.ModelCPPVoidClass class

Package: RTW

Superclasses: RTW.ModelCPPClass

Control C++ class interfaces for models using `void-void` style step method

Description

The `ModelCPPVoidClass` class provides objects that describe C++ class interfaces for models using a `void-void` style step method. Use the `attachToModel` method to attach a C++ class interface to a loaded ERT-based Simulink model.

Construction

`RTW.ModelCPPVoidClass`

Create C++ class interface object for configuring model class with `void-void` style step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following method.

`runValidation`

Validate model-specific C++ class interface against Simulink model

Copy Semantics

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

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. Once you validate and apply your changes, you can generate code based on your C++ class interface modifications. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

RTW.ModelCPPVoidClass

Class: RTW.ModelCPPVoidClass

Package: RTW

Create C++ class interface object for configuring model class with `void-void` style step method

Syntax

```
obj = RTW.ModelCPPVoidClass
```

Description

obj = RTW.ModelCPPVoidClass returns a handle, *obj*, to a newly created object of class RTW.ModelCPPVoidClass.

Output Arguments

<i>obj</i>	Handle to a newly created C++ class interface object for configuring a model class with a <code>void-void</code> style step method. The object has not yet been configured or attached to an ERT-based Simulink model.
------------	--

Alternatives

The **Configure C++ Class Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ class interface dialog box, where you can flexibly control the C++ class interfaces that are generated for your model. See “Customize C++ Class Interfaces Using Graphical Interfaces” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”

- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

RTW.ModelSpecificCPrototype class

Package: RTW

Describe signatures of functions for model

Description

A `ModelSpecificCPrototype` object describes the signatures of the step and initialization functions for a model. You must use this in conjunction with the `attachToModel` method.

Construction

`RTW.ModelSpecificCPrototype`

Create model-specific C prototype object

Methods

`addArgConf`

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

`attachToModel`

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

`getArgCategory`

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

`getArgName`

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

getArgPosition	Get argument position for Simulink model port from model-specific C function prototype
getArgQualifier	Get argument type qualifier for Simulink model port from model-specific C function prototype
getDefaultConf	Get default configuration information for model-specific C function prototype from Simulink model
getFunctionName	Get function name from model-specific C function prototype
getNumArgs	Get number of function arguments from model-specific C function prototype
getPreview	Get model-specific C function prototype code preview
runValidation	Validate model-specific C function prototype against Simulink model
setArgCategory	Set argument category for Simulink model port in model-specific C function prototype
setArgName	Set argument name for Simulink model port in model-specific C function prototype
setArgPosition	Set argument position for Simulink model port in model-specific C function prototype

`setArgQualifier`

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

`setFunctionName`

Set function name in model-specific C function prototype

Copy Semantics

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

Examples

The code below creates a function control object, `a`, and uses it to add argument configuration information to the model.

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

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

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

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

Alternatives

You can create a function control object using the Model Interface dialog box.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

How To

- “Function Prototype Control”

RTW.ModelSpecificCPrototype

Class: RTW.ModelSpecificCPrototype

Package: RTW

Create model-specific C prototype object

Syntax

```
obj = RTW.ModelSpecificCPrototype
```

Description

obj = RTW.ModelSpecificCPrototype creates a handle, *obj*, to an object of class RTW.ModelSpecificCPrototype.

Output Arguments

obj Handle to model specific C prototype object.

Examples

Create a function control object, *a*, and use it to add argument configuration information to the model:

```
% Open the rtdemo_counter model and specify the System Target File
rtdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

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

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

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

Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. See “Configure Function Prototypes Using Graphical Interfaces” in the Embedded Coder documentation.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

How To

- “Function Prototype Control”

rtw.pil.RtIOStreamApplicationFramework

Configure target-side communications

Syntax

```
appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)
```

Description

Constructor	Description
RtIOStreamApplicationFramework	Specify target-specific libraries and source files that are required to build the executable. These libraries and source files must include the device drivers that implement the target-side of the <code>rtiostream</code> communications channel.

Constructor Argument	
componentArgs	An <code>rtw.connectivity.ComponentArgs</code> object.

`appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)` returns an object that provides access to an `RTW.BuildInfo` object containing PIL-specific files (including a PIL main). `rtw.connectivity.MakefileBuilder` combines these files with the PIL component libraries to create the PIL application.

You must make a subclass of `rtw.pil.RtIOStreamApplicationFramework`. In addition:

- Use the `addPILMain` method to specify a `main.c` file, which is required to build the PIL application.
- To the `RTW.BuildInfo` object, add data that is required for the implementation of the `rtiostream` target communications interface, using provided functions.

Required Data	Function for Adding Data
Source file names	“addSourceFiles”
Source file paths	“addSourcePaths”
Include file names	“addIncludeFiles”
Include file paths	“addIncludePaths”
Libraries	“addLinkObjects”
Preprocessor macro definitions	“addDefines”
Compiler options	“addCompileFlags”
Linker options	“addLinkFlags”

The following table describes `rtw.pil.RtIOStreamApplicationFramework` methods.

Method	Syntax and Description
<code>getComponentArgs</code>	<code>componentArgs = appFrameObj.getComponentArgs</code> Returns the <code>rtw.connectivity.ComponentArgs</code> object associated with <code>appFrameObj</code> .
<code>getBuildInfo</code>	<code>buildInfo = appFrameObj.getBuildInfo</code> Returns the <code>RTW.BuildInfo</code> object associated with <code>appFrameObj</code> .
<code>addPILMain</code>	<code>appFrameObj.addPILMain(type)</code> To build the PIL application, a <code>main.c</code> file is required. Use this method to add one of the two provided files to the application framework. <code>type</code> is 'target' or 'host'. To specify a <code>main.c</code> that is adapted for on-target PIL and suitable for most PIL implementations, enter: <code>appFrameObj.addPILMain('target')</code> To specify a <code>main.c</code> that is adapted for host-based PIL, enter: <code>appFrameObj.addPILMain('host')</code>

More About

- “Create PIL Target Connectivity Configuration”
- “Build Information Object”
- `rtwdemo_custom_pil_script`

See Also

`rtw.connectivity.ComponentArgs` | `rtiostream_wrapper`

rtw.pil.RtIOStreamApplicationFramework

Configure target-side communications

Syntax

```
appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)
```

Description

Constructor	Description
<code>RtIOStreamApplicationFramework</code>	Specify target-specific libraries and source files that are required to build the executable file. These libraries and source files must include the device drivers that implement the target-side of the <code>rtiostream</code> communications channel.

Constructor Argument	
<code>componentArgs</code>	An <code>rtw.connectivity.ComponentArgs</code> object.

`appFrameObj = rtw.pil.RtIOStreamApplicationFramework(componentArgs)` returns an object that provides access to an `RTW.BuildInfo` object containing PIL-specific files (including a PIL main). `rtw.connectivity.MakefileBuilder` combines these files with the PIL component libraries to create the PIL application.

You must make a subclass of `rtw.pil.RtIOStreamApplicationFramework`. In addition:

- Use the `addPILMain` method to specify a `main.c` file, which is required to build the PIL application.
- Use the supplied functions to add data to the `RTW.BuildInfo` object, which is required for the implementation of the `rtiostream` target communications interface.

Required Data	Function for Adding Data
Source file names	“addSourceFiles”
Source file paths	“addSourcePaths”
Include file names	“addIncludeFiles”
Include file paths	“addIncludePaths”
Libraries	“addLinkObjects”
Preprocessor macro definitions	“addDefines”
Compiler options	“addCompileFlags”
Linker options	“addLinkFlags”

The following table describes `rtw.pil.RtIOStreamApplicationFramework` methods.

Method	Syntax and Description
<code>getComponentArgs</code>	<p><code>componentArgs = appFrameObj.getComponentArgs</code></p> <p>Returns the <code>rtw.connectivity.ComponentArgs</code> object associated with <code>appFrameObj</code>.</p>
<code>getBuildInfo</code>	<p><code>buildInfo = appFrameObj.getBuildInfo</code></p> <p>Returns the <code>RTW.BuildInfo</code> object associated with <code>appFrameObj</code>.</p>
<code>addPILMain</code>	<p><code>appFrameObj.addPILMain(type)</code></p> <p>To build the PIL application, a <code>main.c</code> file is required. Use this method to add one of the two provided files to the application framework.</p> <p><i>type</i> is 'target' or 'host'.</p> <p>To specify a <code>main.c</code> that is adapted for on-target PIL and suitable for most PIL implementations, enter:</p> <pre>appFrameObj.addPILMain('target')</pre> <p>To specify a <code>main.c</code> that is adapted for host-based PIL, enter:</p> <pre>appFrameObj.addPILMain('host')</pre>

More About

- “Create PIL Target Connectivity Configuration”
- “Build Information Object”

See Also

`rtiostream_wrapper` | `rtw.connectivity.MakefileBuilder` |
`rtw.connectivity.ComponentArgs`

run

Class: `cgv.CGV`

Package: `cgv`

Execute CGV object

Syntax

```
result = cgvObj.run()
```

Description

result = *cgvObj*.run() executes the model once for each input data that you added to the object. *result* is a boolean value that indicates whether the run completed without execution error. *cgvObj* is a handle to a `cgv.CGV` object.

After each execution of the model, the object captures and writes the following metadata to a file in the output folder:

ErrorDetails — If errors occur, the error information.

status — The execution status.

ver — Version information for MathWorks® products.

hostname — Name of computer.

dateTime — Date and time of execution.

warnings — If warnings occur, the warning messages.

username — Name of user.

runtime — The amount of time that lapsed for the execution.

Tips

- Only call `run` once for each `cgv.CGV` object.
- The `cgv.CGV` methods that set up the object are ignored after a call to `run`. See the “`cgv.CGV` class” for details.
- You can call `run` once without first calling “`addInputData (cgv.CGV)`”. However, it is recommended that you first save the required data for execution to a MAT-file,

including the model inputs and parameters. Then use `cgv.CGV.addInputData` to pass the MAT-file to the CGV object before calling `run`.

- The `cgv.CGV` object supports callback functions that you can define and add to the `cgv.CGV` object. These callback functions are called during `cgv.CGV.run()` in the following order:

Callback function	Add to object using...	<code>cgv.CGV.run()</code> executes callback function...
HeaderReportFcn	“addHeaderReportFcn (cgv.CGV)”	Before executing input data in <code>cgv.CGV</code>
PreExecReportFcn	“addPreExecReportFcn (cgv.CGV)”	Before executing each input data file in <code>cgv.CGV</code>
PreExecFcn	“addPreExecFcn (cgv.CGV)”	Before executing each input data file in <code>cgv.CGV</code>
PostExecReportFcn	“addPostExecReportFcn (cgv.CGV)”	After executing each input data file in <code>cgv.CGV</code>
PostExecFcn	“addPostExecFcn (cgv.CGV)”	After executing each input data file in <code>cgv.CGV</code>
TrailerReportFcn	“addTrailerReportFcn (cgv.CGV)”	After the input data is executed in <code>cgv.CGV</code>

How To

- “Verify Numerical Equivalence with CGV”

runValidation

Class: RTW.ModelCPPArgsClass

Package: RTW

Validate model-specific C++ class interface against Simulink model

Syntax

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

Description

[*status*, *msg*] = runValidation(*obj*) runs a validation check of the specified model-specific C++ class interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

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

Output Arguments

status Boolean value; true for a valid configuration, false otherwise.

msg If *status* is false, *msg* contains a string of information describing why the configuration is invalid.

Alternatives

To validate a C++ class interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays status and an explanation of failures. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

runValidation

Class: RTW.ModelCPPVoidClass

Package: RTW

Validate model-specific C++ class interface against Simulink model

Syntax

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

Description

[*status*, *msg*] = runValidation(*obj*) runs a validation check of the specified model-specific C++ class interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getClassInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPVoidClass</code> or <code>obj = RTW.getClassInterfaceSpecification (modelName)</code> .
------------	--

Output Arguments

<i>status</i>	Boolean value; true for a valid configuration, false otherwise.
<i>msg</i>	If <i>status</i> is false, <i>msg</i> contains a string of information describing why the configuration is invalid.

Alternatives

To validate a C++ class interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays status and an explanation of failures. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

runValidation

Class: RTW.ModelSpecificCPrototype

Package: RTW

Validate model-specific C function prototype against Simulink model

Syntax

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

Description

[*status*, *msg*] = runValidation(*obj*) runs a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getFunctionSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

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

Output Arguments

status True for a valid configuration; false otherwise.

msg If *status* is false, *msg* contains a string explaining why the configuration is invalid.

Alternatives

Click the **Validate** button in the Model Interface dialog box to run a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

setArgCategory

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

`setArgCategory(obj, portName, category)` sets the category — 'Value', 'Pointer', or 'Reference' — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPArgsClass</code> or <code>obj = RTW.getClassInterfaceSpecification(<i>modelName</i>)</code> .
<i>portName</i>	String specifying the unqualified name of an inport or outport in your Simulink model.
<i>category</i>	String specifying the argument category — 'Value', 'Pointer', or 'Reference' — to be set for the specified Simulink model port.

Note: If you change the argument category for an outport from 'Pointer' to 'Value', the change causes the argument to move to the first argument position when `attachToModel` or `runValidation` is called.

Alternatives

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

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setArgCategory

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

setArgCategory(*obj*, *portName*, *category*)

Description

setArgCategory(*obj*, *portName*, *category*) sets the category, 'Value' or 'Pointer', of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

<i>obj</i>	Handle to a model-specific C prototype function control object previously returned by <i>obj</i> = RTW.ModelSpecificCPrototype or <i>obj</i> = RTW.getFunctionSpecification(<i>modelName</i>).
<i>portName</i>	String specifying the unqualified name of an inport or outport in your Simulink model.
<i>category</i>	String specifying the argument category, 'Value' or 'Pointer', that you set for the specified Simulink model port.

Note: If you change the argument category for an outport from 'Pointer' to 'Value', it causes the argument to move to the first argument position when you call RTW.ModelSpecificCPrototype.attachToModel or RTW.ModelSpecificCPrototype.runValidation.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument categories. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

setArgName

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

`setArgName(obj, portName, argName)` sets the argument name that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPArgsClass</code> or <code>obj = RTW.getClassInterfaceSpecification(<i>modelName</i>)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.
<i>argName</i>	String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

To set argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you

can display and configure the step method for your model class. In the **I/O arguments step method** view of this dialog box, click the **Get Default Configuration** button to display step method argument names that you can examine and modify. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setArgName

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

`setArgName(obj, portName, argName)`

Description

`setArgName(obj, portName, argName)` sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

<i>obj</i>	Handle to a model-specific C prototype function control object previously returned by <code>obj = RTW.ModelSpecificCPrototype</code> or <code>obj = RTW.getFunctionSpecification(modelName)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.
<i>argName</i>	String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument names. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

setArgPosition

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

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

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ class interface. The specified argument is then moved to the specified position, and other arguments shifted by one position accordingly.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPArgsClass</code> or <code>obj = RTW.getClassInterfaceSpecification(<i>modelName</i>)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.
<i>position</i>	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

To set argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to display step method argument positions that you can examine and modify. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setArgPosition

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

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

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype. The specified argument moves to the specified position, and other arguments shift by one position accordingly.

Input Arguments

<i>obj</i>	Handle to a model-specific C prototype function control object previously returned by <code>obj = RTW.ModelSpecificCPrototype</code> or <code>obj = RTW.getFunctionSpecification(modelName)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.
<i>position</i>	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument position. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

setArgQualifier

Class: RTW.ModelCPPArgsClass

Package: RTW

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

Syntax

`setArgQualifier(obj, portName, qualifier)`

Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — 'none', 'const', 'const *', 'const * const', or 'const &' — of the argument that corresponds to a specified Simulink model inport or output in a specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPArgsClass</code> or <code>obj = RTW.getClassInterfaceSpecification(modelName)</code> .
<i>portName</i>	String specifying the name of an inport or output in your Simulink model.
<i>qualifier</i>	String specifying the argument type qualifier — 'none', 'const', 'const *', 'const * const', or 'const &' — to be set for the specified Simulink model port.

Alternatives

To set argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface**

button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the **I/O arguments step method** view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers that you can examine and modify. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setArgQualifier

Class: RTW.ModelSpecificCPrototype

Package: RTW

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

Syntax

`setArgQualifier(obj, portName, qualifier)`

Description

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

Input Arguments

<i>obj</i>	Handle to a model-specific C prototype function control object previously returned by <code>obj = RTW.ModelSpecificCPrototype</code> or <code>obj = RTW.getFunctionSpecification(modelName)</code> .
<i>portName</i>	String specifying the name of an inport or outport in your Simulink model.
<i>qualifier</i>	String specifying the argument type qualifier — 'none', 'const', 'const *', or 'const * const' — to be set for the specified Simulink model port.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument qualifiers. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

setClassName

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

```
setClassName(obj, className)
```

Description

`setClassName(obj, className)` sets the class name in the specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <i>obj</i> = RTW.ModelCPPArgsClass, <i>obj</i> = RTW.ModelCPPVoidClass, or <i>obj</i> = RTW.getClassInterfaceSpecification(<i>modelName</i>).
<i>className</i>	String specifying a new name for the class described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to

display the model class name, which you can examine and modify. In the `void-void` `step` method view, you can examine and modify the model class name without having to click a button. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setFunctionName

Class: RTW.ModelSpecificCPrototype

Package: RTW

Set function name in model-specific C function prototype

Syntax

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

Description

`setFunctionName(obj, fcnName, fcnType)` sets the step or initialization function name in the specified function control object.

Input Arguments

<i>obj</i>	Handle to a model-specific C prototype function control object previously returned by <code>obj = RTW.ModelSpecificCPrototype</code> or <code>obj = RTW.getFunctionSpecification(modelName)</code> .
<i>fcnName</i>	String specifying a new name for the function described by the function control object. The argument must be a valid C identifier.
<i>fcnType</i>	Optional. String specifying which function to name. Valid strings are 'step' and 'init'. If <i>fcnType</i> is not specified, sets the step function name.

Alternatives

Use the **Initialize function name** and **Step function name** fields in the Model Interface dialog box to specify function names. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

setMode

Class: `cgv.CGV`

Package: `cgv`

Specify mode of execution

Syntax

```
cgvObj.setMode(connectivity)
```

Description

`cgvObj.setMode(connectivity)` specifies the mode of execution for the `cgv.CGV` object, *cgvObj*. The default value for the execution mode is set to either `normal` or `sim`.

Input Arguments

connectivity

Specify mode of execution

Value	Description
<code>sim</code> or <code>normal</code> (default)	Mode of execution is normal simulation.
<code>sil</code>	Mode of execution is SIL.
<code>pil</code>	Mode of execution is PIL.

Examples

After running a `cgv.CGV` object, copy the object. Before rerunning the object, call `setMode` to change the execution mode to `sil` for an existing `cgv.CGV` object.

```
cgvModel = 'rtwdemo_cgv';
cgvObj1 = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

```
cgvObj1.run();  
cgvObj2 = cgvObj1.copySetup()  
cgvObj2.setMode('sil');  
cgvObj2.run();
```

See Also

`cgv.CGV.run` | `cgv.CGV.copySetup`

How To

- “Verify Numerical Equivalence with CGV”

setNameSpace

Set namespace for C++ function entry in code replacement table

Syntax

```
setNameSpace(hEntry, nameSpace)
```

Arguments

hEntry

Handle to a code replacement function entry previously returned by one of the following:

- *hEntry* = RTW.Tf1CFunctionEntry
- *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from RTW.Tf1CFunctionEntry
- A call to the registerCPPFunctionEntry function

nameSpace

String specifying the namespace in which the implementation function for the C++ function entry is defined.

Description

The setNameSpace function specifies the namespace for a C++ function entry in a code replacement table. During code generation, if the function entry is matched, the software emits the namespace in the generated function code (for example, `std::sin(tf1_cpp_U.In1)`).

If you created the function entry using *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = *MyCustomFunctionEntry* (that is, not using registerCPPFunctionEntry), then, before calling the setNameSpace function, you must enable C++ support for the function entry by calling the enableCPP function.

Examples

In the following example, the `setNameSpace` function is used to set the namespace for the `sin` implementation function to `std`.

```
fcn_entry = RTW.Tf1CFunctionEntry;
fcn_entry.setTf1CFunctionEntryParameters( ...
    'Key', 'sin', ...
    'Priority', 100, ...
    'ImplementationName', 'sin', ...
    'ImplementationHeaderFile', 'cmath' );
fcn_entry.enableCPP();
fcn_entry.setNameSpace('std');
```

More About

- “Define Code Replacement Mappings”

See Also

`enableCPP` | `registerCPPFunctionEntry`

setNamespace

Class: RTW.ModelCPPClass

Package: RTW

Set namespace in model-specific C++ class interface

Syntax

```
setNamespace(obj, nsName)
```

Description

`setNamespace(obj, nsName)` sets the namespace in the specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <code>obj = RTW.ModelCPPArgsClass</code> , <code>obj = RTW.ModelCPPVoidClass</code> , or <code>obj = RTW.getClassInterfaceSpecification(<i>modelName</i>)</code> .
<i>nsName</i>	String specifying a namespace for the class described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the model namespace in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the namespace for your model class. In the **I/O arguments step** method view of this dialog box, click the **Get Default Configuration** button to display

the model namespace, which you can examine and modify. In the `void-void step method` view, you can examine and modify the model namespace without having to click a button. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setObjectiveName

Class: rtw.codegenObjectives.Objective

Package: rtw.codegenObjectives

Specify objective name

Syntax

```
setObjectiveName(obj, objName)
```

Description

`setObjectiveName(obj, objName)` specifies a name for the objective. The Configuration Set Objectives dialog box displays the name of the objective.

Input Arguments

<i>obj</i>	Handle to a code generation objective object previously created.
<i>objName</i>	Optional string that indicates the name of the objective. If you do not specify an objective name, the Configuration Set Objectives dialog box displays the objective ID for the objective name.

Examples

Name the objective Reduce RAM Example:

```
setObjectiveName(obj, 'Reduce RAM Example');
```

How To

- “Create Custom Objectives”

setOutputDir

Class: `cgv.CGV`

Package: `cgv`

Specify folder

Syntax

```
cgvObj.setOutputDir('path')  
cgvObj.setOutputDir('path', 'overwrite', 'on')
```

Description

`cgvObj.setOutputDir('path')` is an optional method that specifies a location where the object writes the output and metadata files for execution. `cgvObj` is a handle to a `cgv.CGV` object. `path` is the absolute or relative path to the folder. If the path does not exist, the object attempts to create the folder. If you do not call `setOutputDir`, the object uses the current working folder.

`cgvObj.setOutputDir('path', 'overwrite', 'on')` includes the property and value pair to allow read-only files in the working directory to be overwritten. The default value for 'overwrite' is 'off'.

How To

- “Verify Numerical Equivalence with CGV”

setOutputFile

Class: `cgv.CGV`

Package: `cgv`

Specify output data file name

Syntax

```
cgvObj.setOutputFile(InputIndex,OutputFile)
```

Description

`cgvObj.setOutputFile(InputIndex,OutputFile)` is an optional method that changes the default file name for the output data. `cgvObj` is a handle to a `cgv.CGV` object. *InputIndex* is a unique numeric identifier that specifies which output data to write to the file. The *InputIndex* is associated with specific input data. *OutputFile* is the name of the file, with or without the `.mat` extension.

How To

- “Verify Numerical Equivalence with CGV”

setReservedIdentifiers

Register reserved identifiers to be associated with library

Syntax

```
setReservedIdentifiers(hTable, ids)
```

Arguments

hTable

Handle to a code replacement table previously returned by *hTable* = RTW.Tf1Table.

ids

Structure specifying reserved keywords to be registered for a library. The structure must contain the following:

- **LibraryName** element, a string that specifies 'ANSI_C', 'ISO_C', 'GNU', or a code replacement library name.
- **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_C';  
d{1}.HeaderInfos{1}.HeaderName = 'math.h';  
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

Description

In a code replacement 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 code replacement table. One set of reserved identifiers can be associated with a code replacement library, while the other three (if present) must be associated with libraries named `ANSI_C`, `ISO_C`, or `GNU`.

For information about generating a list of reserved identifiers for the code replacement library that you are using to generate code, see “Reserved Identifiers and Code Replacement”.

Examples

In the following example, `setReservedIdentifiers` is used to register four reserved identifier structures, for `'ANSI_C'`, `'ISO_C'`, `'GNU'`, and `'My Custom CRL'`, respectively.

```
hLib = RTW.Tf1Table;

% Create and register CRL entries here

.
.
.

% Create and register reserved identifiers
d{1}.LibraryName = 'ANSI_C';
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_C';
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';
d{3}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{4}.LibraryName = 'My Custom CRL';
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);
```

More About

- “Reserved Identifiers and Code Replacement”

setStepMethodName

Class: RTW.ModelCPPClass

Package: RTW

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

Syntax

setStepMethodName(*obj*, *fcnName*)

Description

setStepMethodName(*obj*, *fcnName*) sets the step method name in the specified model-specific C++ class interface.

Input Arguments

<i>obj</i>	Handle to a model-specific C++ class interface control object, such as a handle previously returned by <i>obj</i> = RTW.ModelCPPArgsClass, <i>obj</i> = RTW.ModelCPPVoidClass, or <i>obj</i> = RTW.getClassInterfaceSpecification(<i>modelName</i>).
<i>fcnName</i>	String specifying a new name for the step method described by the specified model-specific C++ class interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Class Interface** button. This button launches the Configure C++ class interface dialog box, where you can display and configure the step method for your model class. In the I/O arguments step method view of this dialog box, click the **Get Default Configuration** button to

display the step method name, which you can examine and modify. In the `void-void` step method view, you can examine and modify the step method name without having to click a button. For more information, see “Configure Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Customize C++ Class Interfaces Programmatically”
- “Configure Step Method for Model Class”
- “C++ Class Interface Control”

setTf1CFunctionEntryParameters

Set specified parameters for function entry in code replacement table

Syntax

```
setTf1CFunctionEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a code replacement function entry previously returned by *hEntry* = RTW.Tf1CFunctionEntry or *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from RTW.Tf1CFunctionEntry.

varargin

Parameter/value pairs for the function entry. See *varargin* Parameters.

varargin Parameters

The following function entry parameters can be specified to the `setTf1CFunctionEntryParameters` function using parameter/value argument pairs. For example,

```
setTf1CFunctionEntryParameters(..., 'Key', 'sqrt', ...);
```

Key

String specifying the name of the function to be replaced. The name must match a function listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

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 “Specify Build Information for Replacement Code”.

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 `''`.

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'`.

RoundingModes

Cell array of strings specifying one or more rounding modes 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'`, and `'RTW_ROUND_UNSPECIFIED'`. The default is `{'RTW_ROUND_UNSPECIFIED'}`.

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 string can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a string or cell array of strings in the MATLAB workspace). 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 string can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a string or cell array of strings in the MATLAB workspace). The default is `' '`.

Note: To supply additional build information for the function entry, you can use code replacement table entry functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalLinkObj`, `addAdditionalLinkObjPath`, `addAdditionalSourceFile`, and `addAdditionalSourcePath`, and code replacement table entry properties `AdditionalCompileFlags`, `AdditionalLinkFlags`, and `OtherFiles`. For more information, see “Specify Build Information for Replacement Code”.

AcceptExprInput

Boolean value used to flag the code generator that the implementation function described by this entry should accept expression inputs. The default value is `true` if `ImplType` equals `FCN_IMPL_FUNCT` and `false` if `ImplType` equals `FCN_IMPL_MACRO`.

If the value is `true`, expression inputs are integrated into the generated code in a form similar to the following:

```
rtY.Out1 = mySin(rtU.In1 + rtU.In2);
```

If the value is `false`, a temporary variable is generated for the expression input, as follows:

```
real_T rtb_Sum;

rtb_Sum = rtU.In1 + rtU.In2;
rtY.Out1 = mySin(rtb_Sum);
```

SideEffects

Boolean value used to flag the code generator that the implementation function described by this entry should not be optimized away. This parameter applies to implementation functions that return `void` but should not be optimized away, such as a `memcpy` implementation or an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`. The default is `false`.

StoreFcnReturnInLocalVar

Boolean value used to flag the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false` (the default), other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. For example, here is an example of code generated with expression folding:

```
void sw_step(void)
{
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=
        smul(ssub(sw_U.In4, sw_U.In5), sw_U.In6)) {
        sw_Y.Out1 = sw_U.In7;
    } else {
        sw_Y.Out1 = sw_U.In8;
    }
}
```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code potentially is easier to understand and debug:

```
void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    .....
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
        sw_Y.Out1 = sw_U.In7;
    } else {
        sw_Y.Out1 = sw_U.In8;
    }
}
```

EntryInfoAlgorithm

String specifying a computation or approximation method, configured for the specified math function, that must be matched in order for function replacement to occur. Code replacement libraries support function replacement based on computation or approximation method for the math functions `rSqrt`, `sin`, `cos`, and `sincos`. The valid arguments for each supported function are:

Function	Argument	Meaning
rSqrt	RTW_DEFAULT	Match the default computation method, Exact
	RTW_NEWTON_RAPHSON	Match the Newton-Raphson computation method
	RTW_UNSPECIFIED	Match a computation method
sin cos sincos	RTW_CORDIC	Match the CORDIC approximation method
	RTW_DEFAULT	Match the default approximation method, None
	RTW_UNSPECIFIED	Match an approximation method

Description

The `setTf1CFunctionEntryParameters` function sets specified parameters for a function entry in a code replacement table.

Examples

In the following example, the `setTf1CFunctionEntryParameters` function is used to set specified parameters for a code replacement function entry for `sqrt`.

```
fcn_entry = RTW.Tf1CFunctionEntry;
fcn_entry.setTf1CFunctionEntryParameters( ...
    'Key', 'sqrt', ...
    'Priority', 100, ...
    'ImplementationName', 'sqrt', ...
    'ImplementationHeaderFile', '<math.h>' );
```

More About

- “Define Code Replacement Mappings”

setTf1COperationEntryParameters

Set specified parameters for operator entry in code replacement table

Syntax

```
setTf1COperationEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a code replacement table entry previously returned by one of the following class instantiations:

<code><i>hEntry</i> = RTW.Tf1COperationEntry;</code>	Supports operator replacement.
<code><i>hEntry</i> = RTW.Tf1COperationEntry-Generator;</code>	Provides parameters for fixed-point addition and subtraction that are not available in RTW.Tf1COperationEntry (SlopesMustBeTheSame and ZeroNetBias).
<code><i>hEntry</i> = RTW.Tf1COperationEntry-Generator_NetSlope;</code>	Provides net slope parameters for fixed-point multiplication and division that are not available in RTW.Tf1COperationEntry (NetSlopeAdjustmentFactor and NetFixedExponent).
<code><i>hEntry</i> = RTW.Tf1BlasEntry-Generator;</code>	Supports replacement of nonscalar operators with MathWorks BLAS functions.
<code><i>hEntry</i> = RTW.Tf1CBlasEntry-Generator;</code>	Supports replacement of nonscalar operators with ANSI/ISO [®] C BLAS functions.
<code><i>hEntry</i> = MyCustomOperationEntry;</code> (where <i>MyCustomOperationEntry</i> is a class derived from RTW.Tf1COperationEntry)	Supports operator replacement using custom code replacement table entries.

Note: If you want to specify `SlopesMustBeTheSame` or `MustHaveZeroNetBias` for your operator entry, instantiate your table entry using *hEntry*

= RTW.Tf1COperationEntryGenerator rather than *hEntry* = RTW.Tf1COperationEntry. If you want to use NetSlopeAdjustmentFactor and NetFixedExponent, instantiate your table entry using *hEntry* = RTW.Tf1COperationEntryGenerator_NetSlope.

varargin

Parameter/value pairs for the operator entry. See varargin Parameters.

varargin Parameters

The following operator entry parameters can be specified to the setTf1COperationEntryParameters function using parameter/value argument pairs. For example,

```
setTf1COperationEntryParameters(..., 'Key', 'RTW_OP_ADD', ...);
```

Key

String specifying the operator to be replaced. The string must match an operator key listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”.

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 “Specify Build Information for Replacement Code”.

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.

RoundingModes

Cell array of strings specifying one or more rounding modes supported by the implementation function, among the following: 'RTW_ROUND_FLOOR', 'RTW_ROUND_CEILING', 'RTW_ROUND_ZERO', 'RTW_ROUND_NEAREST', 'RTW_ROUND_NEAREST_ML', 'RTW_ROUND_SIMPLEST', 'RTW_ROUND_CONV', and '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 code replacement request processing must check that slopes of the arguments (input and output) are equal. The default is `false`.

Use this parameter and `MustHaveZeroNetBias` 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.Tf1COperationEntryGenerator` rather than `hEntry = RTW.Tf1COperationEntry`.

MustHaveZeroNetBias

Boolean flag that, when set to `true`, indicates that code replacement request processing must check that net bias of the arguments is zero. The default is `false`.

Use this parameter and `SlopesMustBeTheSame` 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.Tf1COperationEntryGenerator` rather than `hEntry = RTW.Tf1COperationEntry`.

NetSlopeAdjustmentFactor

Floating-point value specifying the slope adjustment factor (F) part of the net slope, $F2^E$, for net slope code replacement entries. The default is `1.0`.

This parameter and `NetFixedExponent` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.Tf1COperationEntryGenerator_NetSlope` rather than `hEntry = RTW.Tf1COperationEntry`.

NetFixedExponent

Floating-point value specifying the fixed exponent (E) part of the net slope, $F2^E$, for net slope code replacement entries. For example, `-3.0`. The default is `0`.

This parameter and `NetsSlopeAdjustmentFactor` 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.Tf1COperationEntryGenerator_NetSlope` rather than `hEntry = RTW.Tf1COperationEntry`.

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 string can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a string or cell array of strings in the MATLAB workspace). 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 string can include tokens (for example, `$myfolder$`, where `myfolder` is a variable defined as a string or cell array of strings in the MATLAB workspace). The default is `''`.

Note: To supply additional build information for the operator entry, you can use code replacement table entry functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalLinkObj`, `addAdditionalLinkObjPath`, `addAdditionalSourceFile`, and

`addAdditionalSourcePath`, and code replacement table entry properties `AdditionalCompileFlags`, `AdditionalLinkFlags`, and `OtherFiles`. For more information, see “Specify Build Information for Replacement Code”.

AcceptExprInput

Boolean value used to flag the code generator that the implementation function described by this entry should accept expression inputs. If the value is `true` (the default), expression inputs are integrated into the generated code in a form similar to the following:

```
rtY.Out1 = u8_add_u8_u8(u8_add_u8_u8(rtU.In1, rtU.In2), rtU.In3);
```

If the value is `false`, a temporary variable is generated for the expression input, as follows:

```
uint8_T tempVar;  
  
tempVar = u8_add_u8_u8(rtU.In1, rtU.In2);  
rtY.Out1 = u8_add_u8_u8(tempVar, rtU.In3);
```

SideEffects

Boolean value used to flag the code generator that the implementation function described by this entry should not be optimized away. This parameter applies to implementation functions that return `void` but should not be optimized away, such as an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`. The default is `false`.

StoreFcnReturnInLocalVar

Boolean value used to flag the code generator that the return value of the implementation function described by this entry must be stored in a local variable regardless of other expression folding settings. If the value is `false` (the default), other expression folding settings determine whether the return value is folded. Storing function returns in a local variable can increase the clarity of generated code. For example, here is an example of code generated with expression folding:

```
void sw_step(void)  
{  
    if (ssub(sadd(sw_U.In1, sw_U.In2), sw_U.In3) <=  
        smul(ssub(sw_U.In4, sw_U.In5), sw_U.In6)) {  
        sw_Y.Out1 = sw_U.In7;  
    } else {  
        sw_Y.Out1 = sw_U.In8;  
    }  
}
```

```

    }
}

```

With `StoreFcnReturnInLocalVar` set to `true`, the generated code potentially is easier to understand and debug:

```

void sw_step(void)
{
    real32_T rtb_Switch;
    real32_T hoistedExpr;
    .....
    rtb_Switch = sadd(sw_U.In1, sw_U.In2);
    rtb_Switch = ssub(rtb_Switch, sw_U.In3);
    hoistedExpr = ssub(sw_U.In4, sw_U.In5);
    hoistedExpr = smul(hoistedExpr, sw_U.In6);
    if (rtb_Switch <= hoistedExpr) {
        sw_Y.Out1 = sw_U.In7;
    } else {
        sw_Y.Out1 = sw_U.In8;
    }
}

```

EntryInfoAlgorithm

String specifying an algorithm, configured for an operation, that must be matched for operator replacement to occur. Valid arguments for relevant operators are:

Operator	Argument	Meaning
RTW_OP_ADD RTW_OP_MINUS	RTW_CAST_BEFORE_OP	Match a cast-before-operation algorithm, which computes the result, and then type casts the result to the output type
	RTW_CAST_AFTER_OP	Match a cast-after-operation algorithm, which type casts input values to the output type prior to computing the result

Description

The `setTf1COperationEntryParameters` function sets specified parameters for an operator entry in a code replacement table.

Examples

In the following example, the `setTf1COperationEntryParameters` function sets parameters for a code replacement operator entry for `uint8` addition that matches a cast-after-sum algorithm.

```
op_entry = RTW.Tf1COperationEntry;  
op_entry.setTf1COperationEntryParameters( ...  
    'Key', 'RTW_OP_ADD', ...  
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...  
    'Priority', 90, ...  
    'SaturationMode', 'RTW_SATURATE_UNSPECIFIED', ...  
    'RoundingModes', {'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 `setTf1COperationEntryParameters` function is used to set parameters for a code replacement operator entry for fixed-point `int16` division. The table entry specifies a net 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.Tf1COperationEntryGenerator_NetSlope;  
op_entry.setTf1COperationEntryParameters( ...  
    'Key', 'RTW_OP_DIV', ...  
    'Priority', 90, ...  
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...  
    'RoundingModes', {'RTW_ROUND_CEILING'}, ...  
    'NetSlopeAdjustmentFactor', 1.0, ...  
    'NetFixedExponent', 0.0, ...  
    'ImplementationName', 's16_div_s16_s16', ...  
    'ImplementationHeaderFile', 's16_div_s16_s16.h', ...  
    'ImplementationSourceFile', 's16_div_s16_s16.c');
```

In the following example, the `setTf1COperationEntryParameters` function is used to set parameters for a code replacement operator entry for fixed-point `uint16` addition that matches a cast-after-sum algorithm. 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.Tf1COperationEntryGenerator;  
op_entry.setTf1COperationEntryParameters( ...  
    'Key', 'RTW_OP_ADD', ...  
    'EntryInfoAlgorithm', 'RTW_CAST_AFTER_OP', ...  
    'Priority', 90, ...  
    'SaturationMode', 'RTW_WRAP_ON_OVERFLOW', ...  
    'RoundingModes', {'RTW_ROUND_UNSPECIFIED'}, ...  
    'SlopesMustBeTheSame', true, ...  
    'MustHaveZeroNetBias', true, ...  
    'ImplementationName', 'u16_add_SameSlopeZeroBias', ...
```

```
'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...  
'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');
```

More About

- “Define Code Replacement Mappings”
- “Scalar Operator Code Replacement”
- “Addition and Subtraction Operator Code Replacement”
- “Small Matrix Operation to Processor Code Replacement”

setTf1CSemaphoreEntryParameters

Set specified parameters for semaphore entry in CRL table

Syntax

```
setTf1CSemaphoreEntryParameters(hEntry, varargin)
```

Arguments

hEntry

Handle to a CRL semaphore entry previously returned by *hEntry* = RTW.Tf1CSemaphoreEntry;

varargin

Parameter/value pairs for the semaphore entry. See *varargin* Parameters.

varargin Parameters

The following semaphore entry parameters can be specified to the `setTf1CSemaphoreEntryParameters` function using parameter/value argument pairs. For example,

```
setTf1CSemaphoreEntryParameters(..., 'Key', 'RTW_SEM_INIT', ...);
```

Key

String specifying the semaphore or mutex function to be replaced. The string must match a key listed in “Code You Can Replace from MATLAB Code” or “Code You Can Replace From Simulink Models”

GenCallback

String specifying `''` or `'RTW.copyFileToBuildDir'`. The default is `''`. If you specify `'RTW.copyFileToBuildDir'`, and if this semaphore entry is matched and used, the function `RTW.copyFileToBuildDir` is called after code generation to copy additional header, source, or object files that you have specified for this semaphore entry to the build directory. For more information, see “Specify Build Information for Replacement Code”.

Priority

Positive integer specifying the semaphore entry's search priority, 0-100, relative to other entries of the same 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 semaphore operation, the implementation with the higher priority will shadow the one with the lower priority.

ImplementationName

String specifying the name of the implementation function, for example, 'mySemCreate'.

ImplementationHeaderFile

String specifying the name of the header file that declares the implementation function, for example, 'mySem.h'. The default is ''.

ImplementationHeaderPath

String specifying the full path to the implementation header file. The string can include tokens (for example, \$myfolder\$, where myfolder is a variable defined as a string or cell array of strings in the MATLAB workspace). The default is ''.

ImplementationSourceFile

String specifying the name of the implementation source file, for example, 'mySem.c'. The default is ''.

ImplementationSourcePath

String specifying the full path to the implementation source file. The string can include tokens (for example, \$myfolder\$, where myfolder is a variable defined as a string or cell array of strings in the MATLAB workspace). The default is ''.

Note: To supply additional build information for the semaphore entry, you can use CRL table entry functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalLinkObj`, `addAdditionalLinkObjPath`, `addAdditionalSourceFile`, and `addAdditionalSourcePath`, and CRL table entry properties `AdditionalCompileFlags`, `AdditionalLinkFlags`, and `OtherFiles`. For more information, see “Specify Build Information for Replacement Code”.

SideEffects

Boolean value used to flag the code generator that the implementation function described by this entry should not be optimized away. This parameter applies to

implementation functions that return `void` but should not be optimized away, such as an implementation function that accesses global memory values. For those implementation functions only, you must include this parameter and specify the value `true`. The default is `false`.

Description

The `setTf1CSemaphoreEntryParameters` function sets specified parameters for a semaphore entry in a CRL table.

Examples

In the following example, the `setTf1CSemaphoreEntryParameters` function is used to set specified parameters for a CRL table entry for a semaphore initialization replacement.

```
sem_entry = RTW.Tf1CSemaphoreEntry;
sem_entry.setTf1CSemaphoreEntryParameters( ...
    'Key', 'RTW_SEM_INIT', ...
    'Priority', 100, ...
    'ImplementationName', 'mySemCreate', ...
    'ImplementationHeaderFile', 'mySem.h', ...
    'ImplementationSourceFile', 'mySem.c', ...
    'GenCallback', 'RTW.copyFileToBuildDir', ...
    'SideEffects', true);
```

More About

- “Semaphore and Mutex Function Replacement”
- “Define Code Replacement Mappings”

coder.MATLABCodeTemplate.setTokenValue

Class: coder.MATLABCodeTemplate

Package: coder

Set value of token for code generation template

Syntax

```
setTokenValue(tokenName,tokenValue)
```

Description

`setTokenValue(tokenName,tokenValue)` sets the value of a token for a code generation template.

Input Arguments

tokenName

The name of the token

Default:

tokenValue

The value of the token

Default: empty

Examples

Create a `MATLABCodeTemplate` object from a custom template. Set the value for a custom token in the template.

```
newObj = coder.MATLABCodeTemplate('myCGTFile');
```

```
% Create a MATLABCodeTemplate object from a custom template file
newObj.setTokenValue('myCustomToken', 'myValue');
% Set the value of a custom token in the file
newObj.getTokenValue('myCustomToken')
% Check value of the custom token
```

See Also

`coder.MATLABCodeTemplate.getTokenValue` |
`coder.MATLABCodeTemplate.getCurrentTokens` |
`coder.MATLABCodeTemplate.emitSection`

Related Examples

- “Generate Custom File and Function Banners for C and C++ Code”

More About

- “Code Generation Template (CGT) Files for MATLAB”

run

Execute program loaded on processor

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

runopt string	Description
'run'	Executes the run and waits to confirm that the processor is running, and then returns to MATLAB.
'runtohalt'	Executes the run but then waits until the processor halts before returning. The halt can be the result of the PC

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

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

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

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

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

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

See Also

halt | load | “reset”

save

Save file

Syntax

```
save(IDE_Obj, filename, filetype)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

Use `save(IDE_Obj, filename, filetype)` to save open files in the IDE project.

The *filename* argument defines the name of the file to save. When entering the file name, include the file extension.

The optional *filetype* argument defines the type of file to save. If you omit the *filetype* argument, *filetype* defaults to 'project'. Except with VisualDSP++ IDE, 'project' is the only supported option. Therefore, you can omit the *filetype* argument in most cases.

	CCS IDE	VisualDSP++ IDE
'project'	Yes	Yes
'projectgroup'	No	Yes

Note: The open method does not support the 'text' argument.

Examples

To save the project files:

```
save(IDE_Obj, 'all')
```

To save the myproject project:

```
save(IDE_Obj, 'myproject')
```

To save the active project:

```
save(IDE_Obj, [])
```

For VisualDSP++ IDE, to save the projects in the project groups:

```
save(IDE_Obj, 'all', 'projectgroup')
```

For VisualDSP++ IDE, to save the myg.dpg project group:

```
save(IDE_Obj, 'myg.dpg', 'projectgroup')
```

For VisualDSP++ IDE, to save the active project in the project groups:

```
save(IDE_Obj, [], 'projectgroup')
```

See Also

adivdsp | close | load

setbuilddopt

Set active configuration build options

Syntax

```
setbuilddopt(IDE_Obj,tool,ostr)  
setbuilddopt(IDE_Obj,file,ostr)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

Use `setbuilddopt(IDE_Obj,tool,ostr)` to set the build options for a specific build tool in the current configuration. This replaces the switch settings that are applied when you invoke the command line `tool`. For example, a build tool could be a compiler, linker or assembler. To define the `tool` argument, first use the `getbuilddopt` command to read a list of defined build tools.

If the VisualDSP++ and Code Composer Studio IDEs do not recognize the `ostr` argument, `setbuilddopt` sets the switch settings to the default values for the build tool specified by `tool`.

Use `setbuilddopt(IDE_Obj,file,ostr)` to configure the build options for a file you specify with the `file` argument. The source file must exist in the active project.

See Also

activate | getbuilddopt

symbol

Program symbol table from IDE

Syntax

```
s = symbol(IDE_Obj)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

`s = symbol(IDE_Obj)` returns the symbol table for the program loaded in the processor associated with the IDE handle object, `IDE_Obj`. The `symbol` method only applies after you load a processor program file. `s` is an array of structures where each row in `s` presents the symbol name and address in the table. Therefore, `s` has two columns; one is the symbol name, and the other is the symbol address and symbol page.

For CCS IDE, this table shows a few possible elements of `s`, and their interpretation.

s Structure Field	Contents of the Specified Field
<code>s(1).name</code>	String reflecting the symbol entry name.
<code>s(1).address(1)</code>	Address or value of symbol entry.
<code>s(1).address(2)</code>	Memory page for the symbol entry. For TI C6xxx processors, the page is 0.

You can use field `address` in `s` as the `address` input argument to `read` and `write`.

If you use `symbol` and the symbol table does not exist, `s` returns empty and you get a warning message.

Symbol tables are a portion of a COFF object file that contains information about the symbols that are defined and used by the file. When you load a program to the processor, the symbol table resides in the IDE. While the IDE may contain more than one symbol table at a time, `symbol` accesses the symbol table belonging to the program you last loaded on the processor.

Examples

Build and load an example program on your processor. Then use `symbol` to return the entries stored in the symbol table in the processor.

```
s = symbol(IDE_Obj);
```

`s` contains the symbols and their addresses, in a structure you can display with the following code:

```
for k = 1:length(s), disp(k), disp(s(k)), end;
```

MATLAB software lists the symbols from the symbol table in a column.

See Also

`load` | `run`

ticcs

Create handle object to interact with Code Composer Studio IDE

Syntax

```
IDE_Obj = ticcs  
IDE_Obj = ticcs('propertyname', 'propertyvalue', ...)
```

Note: The output argument name you provide for `ticcs` cannot begin with an underscore, such as `_IDE_Obj`.

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`IDE_Obj = ticcs` returns a `ticcs` object in `IDE_Obj` that MATLAB software uses to communicate with the default processor. If you do not use input arguments, `ticcs` constructs the object with default values for the properties. the IDE handles the communications between MATLAB software and the selected CPU. When you use the function, `ticcs` starts the IDE if it is not running. If `ticcs` opened an instance of the IDE when you issued the `ticcs` function, the IDE becomes invisible after your coder product creates the new object.

Note: When `ticcs` creates the object `IDE_Obj`, it sets the working folder for the IDE to be the same as your MATLAB Current Folder. When you create files or projects in the IDE, or save files and projects, this working folder affects where you store the files and projects.

Each object that accesses the IDE comprises two objects—a `ticcs` object and an `rtdx` object—that include the following properties.

Object	Property Name	Property	Default	Description
ticcs	'apiversion'	API version	N/A	Defines the API version used to create the link.
	'proctype'	Processor Type	N/A	Specifies the kind of processor on the board.
	'procname'	Processor Name	CPU	Name given to the processor on the board to which this object links.
	'status'	Running	No	Status of the program currently loaded on the processor.
	'boardnum'	Board Number	0	Number that CCS assigns to the board. Used to identify the board.
	'procnum'	Processor number	0	Number the CCS assigns to a processor on a board.
	'timeout'	Default timeout	10.0 s	Specifies how long MATLAB software waits for a response from CCS after issuing a request. This also applies when you try to construct a <code>ticcs</code> object. The create process waits for this timeout period for the connection to the processor to complete. If the timeout period expires, you get an error message that the connection to the processor failed and MATLAB software could not create the <code>ticcs</code> object.
rtdx	'timeout'	Timeout	10.0 s	Specifies how long CCS waits for a response from the

Object	Property Name	Property	Default	Description
				processor after requesting data.
	'numchannels'	Number of open channels	0	The number of open channels using this link.

`IDE_Obj = ticcs('propertyname', 'propertyvalue', ...)` returns a handle in `IDE_Obj` that MATLAB software uses to communicate with the specified processor. CCS handles the communications between the MATLAB environment and the CPU.

MATLAB software treats input parameters to `ticcs` as property definitions. Each property definition consists of a property name/property value pair.

Two properties of the `ticcs` object are read only after you create the object:

- `'boardnum'` — The identifier for the installed board selected from the active boards recognized by CCS. If you have one board, use the default property value 0 to access the board.
- `'procnum'` — The identifier for the processor on the board defined by `boardnum`. On boards with more than one processor, use this value to specify the processor on the board. On boards with one processor, use the default property value 0 to specify the processor.

Given these two properties, the most common forms of the `ticcs` method are

```
IDE_Obj = ticcs('boardnum',value)
IDE_Obj = ticcs('boardnum',value,'procnum',value)
IDE_Obj = ticcs(...,'timeout',value)
```

which specify the board, and processor in the second example, as the processor.

The third example adds the `timeout` input argument and `value` to allow you to specify how long MATLAB software waits for the connection to the processor or the response to a command to return completed.

You do not need to specify the `boardnum` and `procnum` properties when you have one board with one processor installed. The default property values refer to the processor on the board.

Note: Simulators are considered boards. If you defined both boards and simulators in the IDE, specify the `boardnum` and `procnum` properties to connect to specific boards or

simulators. Use `ccsboardinfo` to determine the values for the `boardnum` and `procnum` properties.

Because these properties are read only after you create the handle, you must set these property values as input arguments when you use `ticcs`. You cannot change these values after the handle exists. After you create the handle, use the `get` function to retrieve the `boardnum` and `procnum` property values.

Using `ticcs` with Multiple Processor Boards

When you create `ticcs` objects that access boards that contain more than one processor, such as the OMAP1510 platform, `ticcs` behaves a little differently.

For each of the `ticcs` syntaxes, the result of the method changes in the multiple processor case, as follows.

```
IDE_Obj = ticcs
IDE_Obj = ticcs('propertyname',propertyvalue)
IDE_Obj = ticcs('propertyname',propertyvalue,'propertyname',...
propertyvalue)
```

In the case where you do not specify a board or processor:

```
IDE_Obj = ticcs
Array of TICCS Objects:
API version          : 1.2
Board name           : OMAP 3.0 Platform Simulator [Texas
Instruments]
Board number         : 0
Processor 0 (element 1): TMS470R2127 (MPU, Not Running)
Processor 1 (element 2): TMS320C5500 (DSP, Not Running)
```

Where you choose to identify your processor as an input argument to `ticcs`, for example, when your board contains two processors:

```
IDE_Obj = ticcs('boardnum',2)
Array of TICCS Objects:
API version          : 1.2
Board name           : OMAP 3.0 Platform Simulator [Texas Instruments]
Board number         : 2
Processor 0 (element 1): TMS470R2127 (MPU, Not Running)
Processor 1 (element 2): TMS320C5500 (DSP, Not Running)
```

`IDE_Obj` returns a two element object handle with `IDE_Obj (1)` corresponding to the first processor and `IDE_Obj (2)` corresponding to the second.

You can include both the board number and the processor number in the `ticcs` syntax. For example:

```

IDE_Obj = ticcs('boardnum',2,'procnum',[0 1])
Array of TICCS Objects:
  API version           : 1.2
  Board name           : OMAP 3.0 Platform Simulator [Texas
  Instruments]
  Board number         : 2
  Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
  Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)

```

Enter `procnum` as either a single processor on the board (a single value in the input arguments to specify one processor) or a vector of processor numbers, as shown in the example, to select two or more processors.

Support Coemulation and OMAP

Coemulation, defined by Texas Instruments to mean simultaneous debugging of two or more CPUs, allows you to coordinate your debugging efforts between two or more processors within one device. Efficient development with OMAP™ hardware requires coemulation support. Instead of creating one `IDE_Obj` object when you issue the following command

```
IDE_Obj = ticcs
```

or your hardware that has multiple processors, the resulting `IDE_Obj` object comprises a vector of `IDE_Obj` objects `IDE_Obj(1)`, `IDE_Obj(2)`, and so on, each of which accesses one processor on your device, say an OMAP1510. When your processor has one processor, `IDE_Obj` is a single object. With a multiprocessor board, the `IDE_Obj` object returns the new vector of objects. For example, for board 2 with two processors,

```
IDE_Obj = ticcs
```

returns the following information about the board and processors:

```

IDE_Obj = ticcs('boardnum',2)
Array of TICCS Objects:
  API version           : 1.2
  Board name           : OMAP 3.0 Platform Simulator [Texas
  Instruments]
  Board number         : 2
  Processor 0 (element 1) : TMS470R2127 (MPU, Not Running)
  Processor 1 (element 2) : TMS320C5500 (DSP, Not Running)

```

Checking the existing boards shows that board 2 does have two processors:

```
ccsboardinfo
```

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	OMAP 3.0 Platform Simulator [T ...	0	MPU	TMS470R2x

2	OMAP 3.0 Platform Simulator [T ...	1	DSP	TMS320C550
1	MGS3 Simulator [Texas Instruments]	0	CPU	TMS320C5500
0	ARM925 Simulator [Texas Instru ...	0	CPU	TMS470R2x

Examples

On a system with three boards, where the third board has one processor and the first and second boards have two processors each, the following function:

```
IDE_Obj = ticcs('boardnum',1,'procnum',0);
```

returns an object that accesses the first processor on the second board. Similarly, the function

```
IDE_Obj = ticcs('boardnum',0,'procnum',1);
```

returns an object that refers to the second processor on the first board.

To access the processor on the third board, use

```
IDE_Obj = ticcs('boardnum',2);
```

which sets the default property value `procnum = 0` to connect to the processor on the third board.

```
IDE_Obj = ticcs
TICCS Object:
API version      : 1.2
Processor type   : TMS320C6711
Processor name   : CPU_1
Running?        : No
Board number     : 1
Processor number : 0
Default timeout  : 10.00 secs

RTDX channels    : 0
```

Defined types : Void, Float, Double, Long, Int, Short, Char

See Also

`ccsboardinfo` | `set`

visible

Set whether IDE window appears while IDE runs

Syntax

```
visible(IDE_Obj,state)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

Use `visible(IDE_Obj,state)` to make the IDE visible on the desktop or make it run in the background.

To run the IDE in the background so it is not visible on the desktop, enter '0' for the *state* argument.

To make the IDE visible on your system desktop, enter '1' for the *state* argument.

You can use methods to interact with a IDE handle object, such as `IDE_Obj`, while the IDE is in both states, visible and not visible. You can interact with the IDE GUI while the IDE is visible.

On the Microsoft Windows platform, if you make the IDE visible and look at the Windows Task Manager:

- While the IDE is visible (*state* is 1), the IDE appears on the **Applications** page of Task Manager, and the `IDE_Obj_app.exe` process shows up on the **Processes** page as a running process.

- While the IDE is not visible (*state* is 0), the IDE disappears from the **Applications** page, but remains on the **Processes** page, with a process ID (PID), using CPU and memory resources.

Examples

In MATLAB, use the constructor function to create a IDE handle object for your IDE. The constructor function creates a handle, such as `IDE_Obj`, and starts the IDE.

To get the visibility status of `IDE_Obj`, enter:

```
isvisible(IDE_Obj)

ans =
     0
```

Now, change the visibility of the IDE to 1, and check its visibility again.

```
visible(IDE_Obj,1)
isvisible(IDE_Obj)

ans =
     1
```

If you close MATLAB software while the IDE is not visible, the IDE remains running in the background. To close it, perform either of the following tasks:

- Start MATLAB software. Create a link to the IDE. Use the new link to make the IDE visible. Close the IDE.
- Open Microsoft Windows Task Manager. Click **Processes**. Find and highlight `IDE_Obj_app.exe`. Click **End Task**.

See Also

`isvisible` | `load`

write

Write data to processor memory block

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

The `address` argument is a decimal or hexadecimal representation of a memory address in the processor. The full memory address consist of two parts: the start address and the

memory type. The memory type value can be explicitly defined using a numeric vector representation of the address.

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

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

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

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

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

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

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

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

`mem = write(...,datatype)` where the *datatype* argument defines the interpretation of the raw values written to DSP memory. The *datatype* argument specifies the data

format of the raw memory image. The data is written starting from `address` without regard to data type alignment boundaries in the DSP. The byte ordering of the data type is automatically applied. The following MATLAB data types are supported.

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

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

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

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

Using write with VisualDSP++ IDE

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

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

Examples

Example with VisualDSP++ IDE

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

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

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

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

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

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

See Also

hex2dec | read

writemsg

Write messages to specified RTDX channel

Note: Support for writemsg on C5000 processors will be removed in a future version.

Syntax

```
data = writemsg(rx,channelname,data)
data = writemsg(rx,channelname,data,timeout)
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`data = writemsg(rx,channelname,data)` writes `data` to a channel associated with `rx`. `channelname` identifies the channel queue, which you must configure for write access beforehand. The messages must be the same type for a single write operation. `writemsg` takes the elements of matrix `data` in column-major order.

In `data = writemsg(rx,channelname,data,timeout)`, the optional argument, `timeout`, limits the time `writemsg` spends transferring messages from the processor. `timeout` is the number of seconds allowed to complete the write operation. You can use `timeout` limit prolonged data transfer operations. If you omit `timeout`, `writemsg` applies the global timeout period defined for the IDE handle object `IDE_Obj`.

`writemsg` supports the following data types: `uint8`, `int16`, `int32`, `single`, and `double`.

Examples

After you load a program to your processor, configure a link in RTDX for write access and use `writemsg` to write data to the processor. Recall that the program loaded on the processor must define `ichannel` and the channel must be configured for write access.

```
IDE_Obj = ticcs;
rx = rtdx(IDE_Obj);
open(rx,'ichannel','w'); % Could use rx.open('ichannel','w')
enable(rx,'ichannel');
inputdata(1:25);
writemsg(rx,'ichannel',int16(inputdata));
```

As a further illustration, the following code snippet writes the messages in matrix `indata` to the write-enabled channel specified by `ichan`. The code in this example processes only when `ichan` is defined by the program on the processor and enabled for write access.

```
indata = [1 4 7; 2 5 8; 3 6 9];
writemsg(rtdx(IDE_Obj),'ichan',indata);
```

The matrix `indata` is written by column to `ichan`. The preceding function syntax is equivalent to

```
writemsg(rtdx(IDE_Obj),'ichan',[1:9]);
```

See Also

`readmat` | `readmsg` | `write`

xmakefilesetup

Configure your coder product to generate makefiles

Syntax

```
xmakefilesetup
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3
- Texas Instruments Code Composer Studio v4
- Texas Instruments Code Composer Studio v5

Description

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

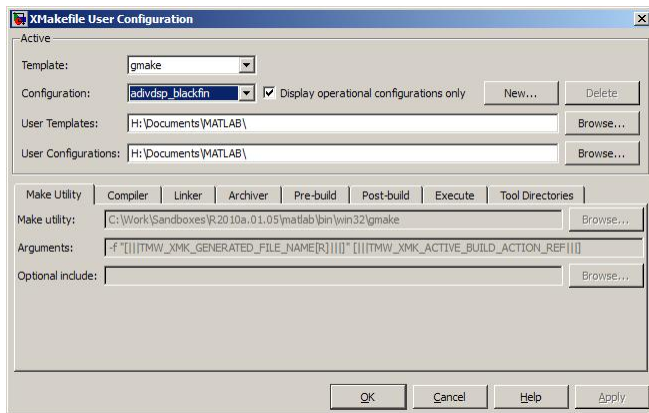
Enter `xmakefilesetup` at the MATLAB command line to configure how to generate makefiles.

Use this function:

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

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

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



See Also

“Build format” on page 3-86 | “Build action” on page 3-87

Blocks — Alphabetical List

Byte Pack

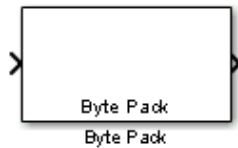
Convert input signals to `uint8` vector

Library

Embedded Coder/ Embedded Targets/ Host Communication

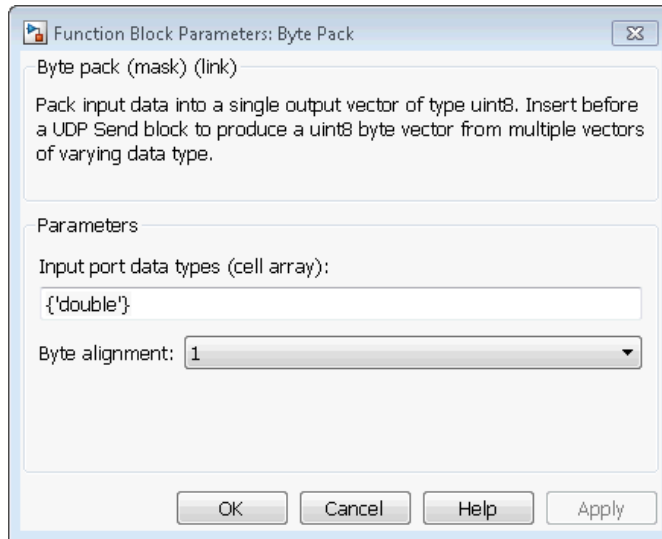
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Target Communication

Description



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

Dialog Box



Input port data types (cell array)

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

Byte alignment

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

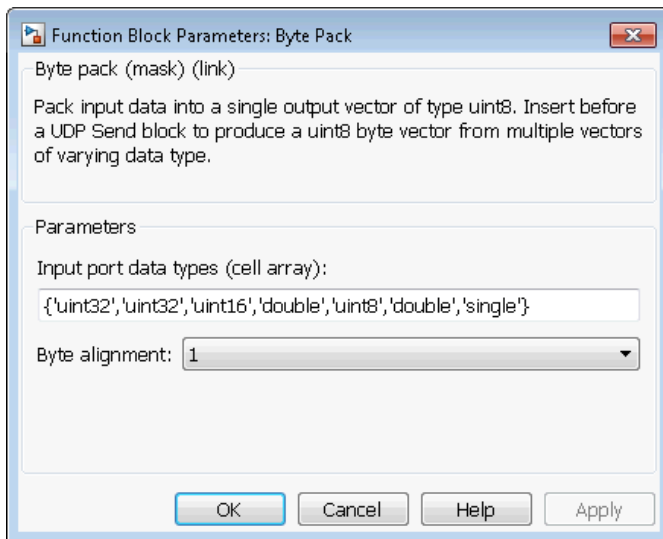
Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to 1. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm states that each element in the output vector begins on a byte boundary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

Selecting 1 for **Byte alignment** provides the tightest packing, without holes between data types in the various combinations of data types and signals.

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

Example

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



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

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

The example shows the following data types:

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

When the signals are scalar values (not matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

See Also

“Byte Reversal”, “Byte Unpack”

Byte Reversal

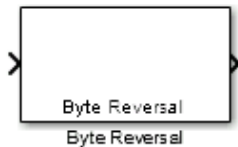
Reverse order of bytes in input word

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Target Communication

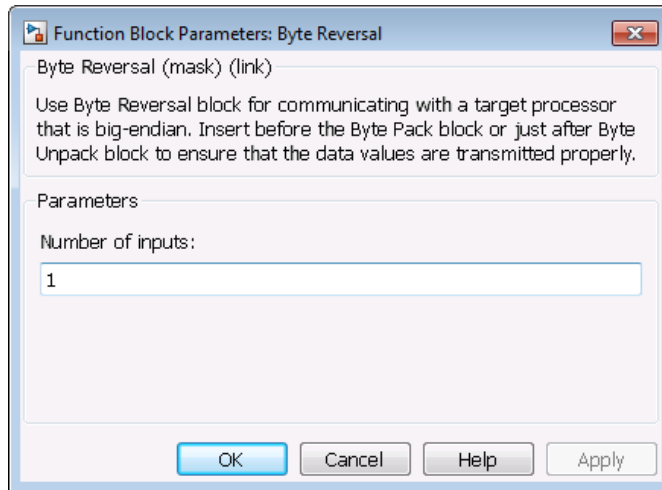
Description



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

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

Dialog Box



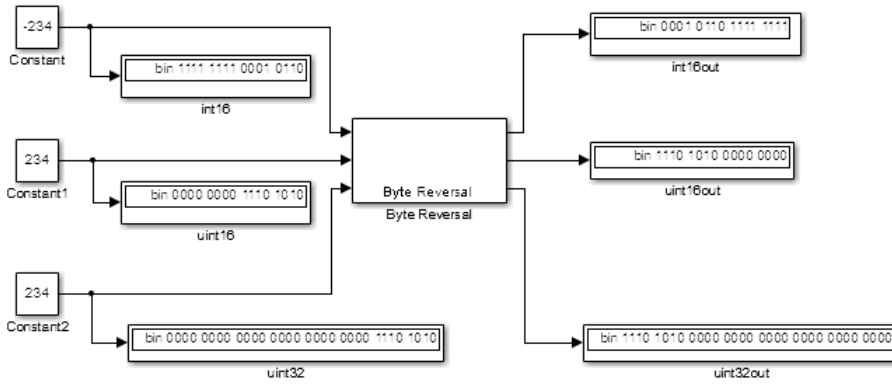
Number of inputs

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

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

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

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



See Also

“Byte Pack”, “Byte Unpack”

Byte Unpack

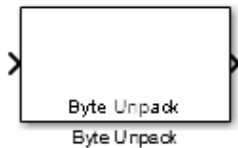
Unpack UDP `uint8` input vector into Simulink data type values

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Target Communication

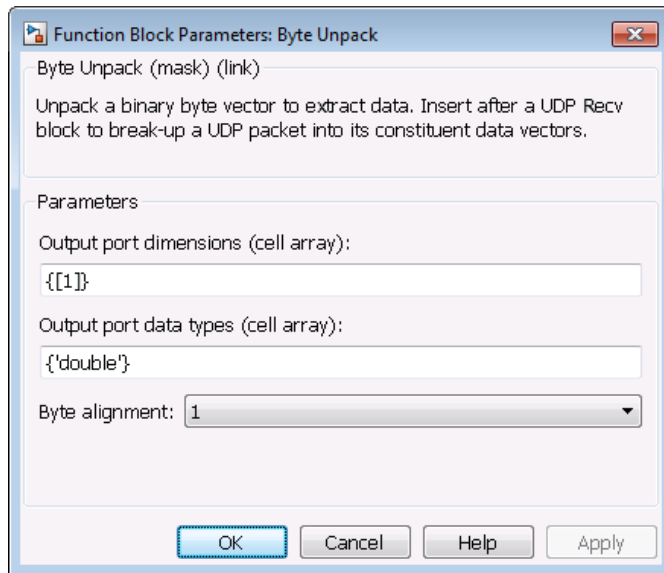
Description



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

The block supports all Simulink data types.

Dialog Box



Output port dimensions (cell array)

Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding “Byte Pack” block in the model. Entering one value means that the block applies that dimension to all data types.

Output port data types (cell array)

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

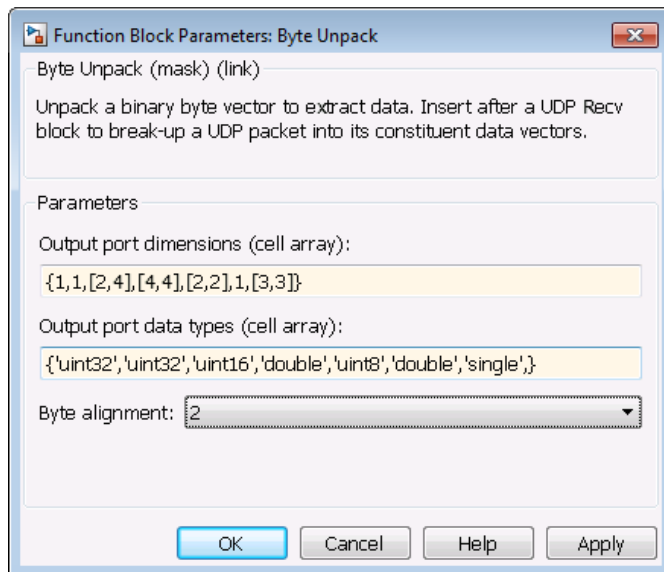
Byte Alignment

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

Example

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

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



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

See Also

“Byte Pack”, “Byte Reversal”

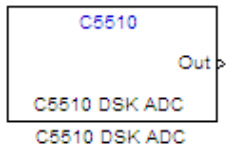
C5510 DSK ADC

Configure AIC23 and peripherals to collect data from analog jacks and output digital data

Library

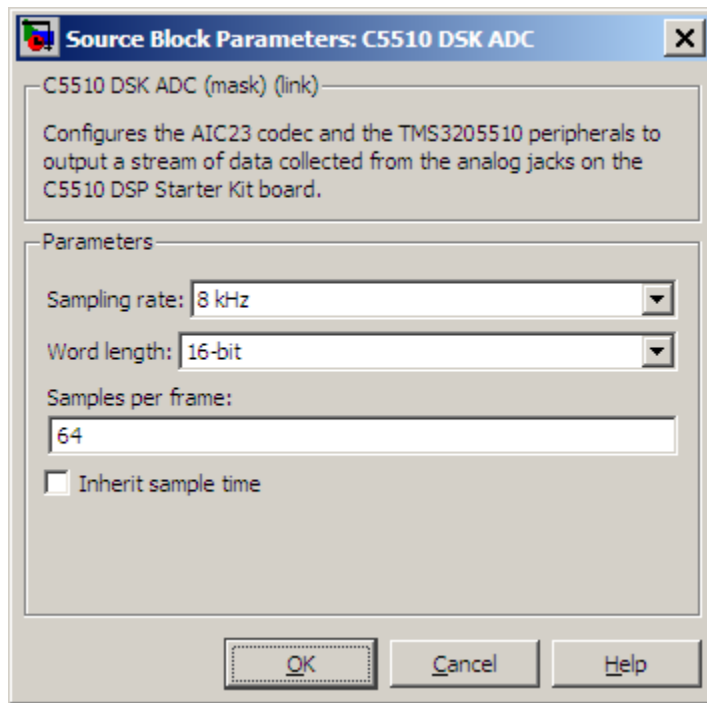
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ C5510 DSK

Description



Configures the AIC23 codec and the TMS320C5510 peripherals to output a stream of digital data. The block collects this data from the analog jacks on the C5510 DSP Starter Kit board.

Dialog Box



Sampling rate

Set the rate at which the analog-to-digital converter samples the analog input. A higher rate increases the resolution of the data the ADC outputs.

Word length

Set the number of data bits the ADC creates for each sample. Increasing the word length increases the accuracy of the data in each sample. If your model also contains a DAC block, set the word length in the DAC block to match that of the ADC block.

Samples per frame

Set the number of samples the ADC buffers internally before it sends the digitized signals, as a frame vector, to the next block in the model. This value defaults to 64 samples per frame. The frame rate depends on the sample rate and frame size. Thus, if you set **Sampling Rate** to 8 kHz, and **Samples per frame** to 32, the resulting frame rate is 250 frames per second ($8000/32 = 250$).

Inherit sample time

Select whether the block inherits the sample time from the model base rate or from the Simulink base rate. You can locate the Simulink base rate in the Solver options in Configuration Parameters. Selecting Inherit sample time directs the block to use the specified rate in model configuration. Entering -1 configures the block to accept the sample rate from the upstream HWI, Task, or Triggered Task blocks.

See Also

C5510 DSK DAC

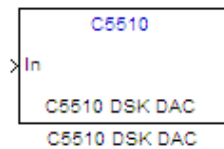
C5510 DSK DAC

Configure AIC23 codec and peripherals to send data stream to output jack

Library

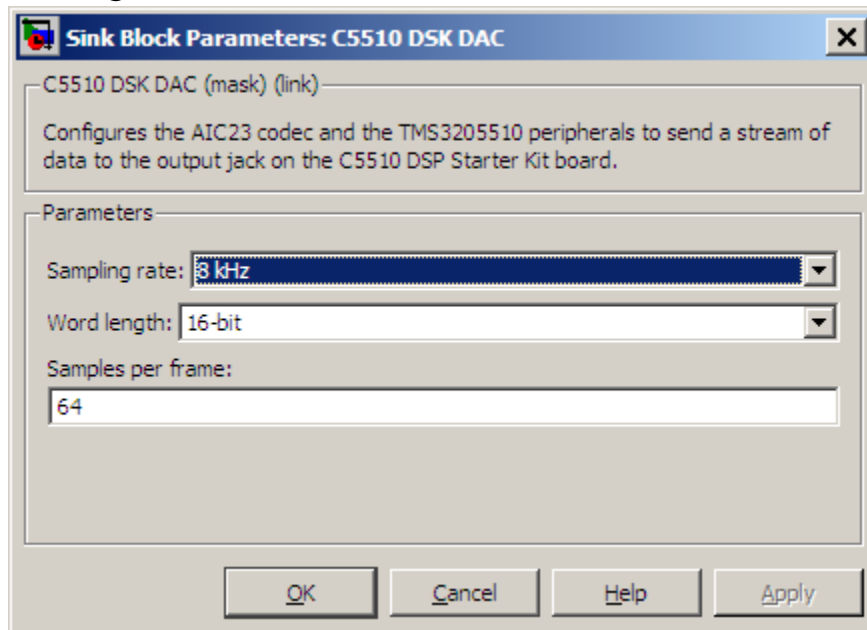
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ C5510 DSK

Description



Configures the AIC23 codec and the TMS3205510 peripherals to send a stream of data to the output jack on the C5510 DSP Starter Kit board.

Dialog Box



Sampling Rate

Set the rate at which the digital-to-analog converter receives each data sample. If your model contains an ADC block, set this value to match the sampling rate of the ADC block.

Word length

Set the number of bits in each data input sample the DAC. If your model also contains an ADC block, set the word length in the DAC block to match that of the ADC block. If you enter the incorrect value for this parameter, the DAC cannot generate an analog output that corresponds to the data it receives.

Samples per frame

Set the number of samples per data input frame. Match this value with the value of the block creating the data frames. This value defaults to 64 samples per frame.

See Also

C5510 DSK ADC

C5000/C6000 Hardware Interrupt

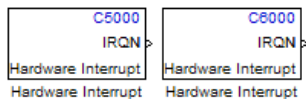
Interrupt Service Routine to handle hardware interrupt on C5000 and C6000 processors

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ Scheduling

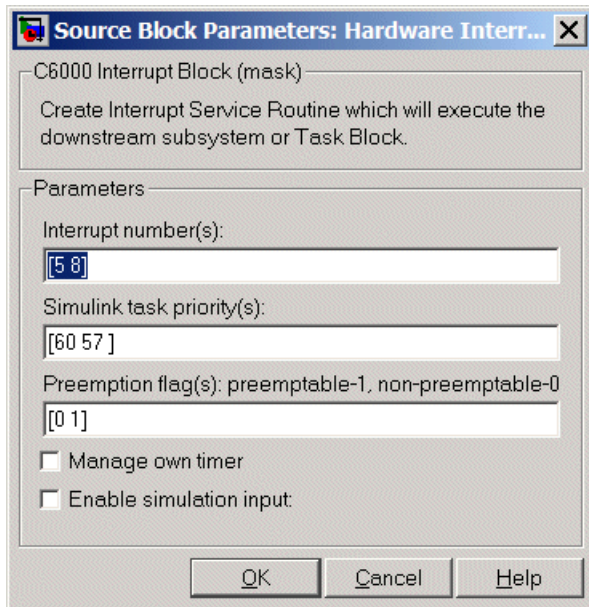
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from the this block or a Task block connected to this block.

Dialog Box



Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The following table provides the valid range for C5xxx and C6xxx processors:

Processor Family	Valid Interrupt Numbers
C5xxx	2, 3, 5-21, 23
C6xxx	4-15

The width of the block output signal corresponds to the number of interrupt numbers specified here. Combined with the **Simulink task priorities** that you enter and the **preemption flag** you enter for each interrupt, these three values define how the code and processor handle interrupts during asynchronous scheduler operations.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink software task priority specifies

the Simulink priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Simulink task priority values are required to generate rate transition code (refer to “Rate Transitions and Asynchronous Blocks”). The task priority values are also required for absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. Typically, you assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags preemptable – 1, non-preemptable – 0

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 5 in **Interrupt numbers** is not preemptible and the priority 8 interrupt can be preempted.

Enable simulation input

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

CAN Pack

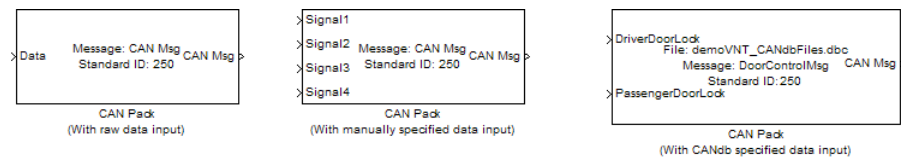
Pack individual signals into CAN message

Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

Description



The CAN Pack block loads signal data into a message at specified intervals during the simulation.

Note: To use this block, you also need a license for Simulink software.

CAN Pack block has one input port by default. The number of block inputs is dynamic and depends on the number of signals you specify for the block. For example, if your block has four signals, it has four block inputs.

This block has one output port, CAN Msg. The CAN Pack block takes the specified input parameters and packs the signals into a message.

Other Supported Features

The CAN Pack block supports:

- The use of Simulink Accelerator™ Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

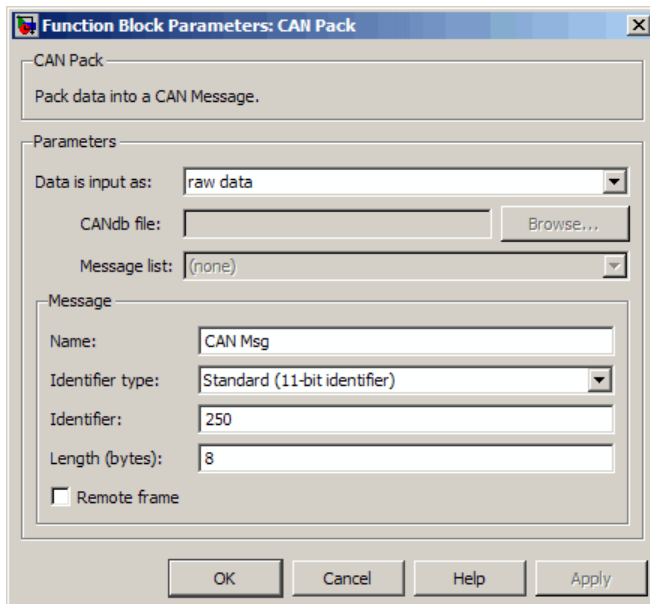
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Simulink Coder to deploy models to targets.

Note: Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32-bits long.

For more information on these features, see the Simulink documentation.

Dialog Box

Use the Function Block Parameters dialog box to select your CAN Pack block parameters.



Parameters

Data is input as

Select your data signal:

- **raw data:** Input data as a uint8 vector array. If you select this option, you only specify the message fields. all other signal parameter fields are unavailable. This option opens only one input port on your block.
- **manually specified signals:** Allows you to specify data signal definitions. If you select this option, use the **Signals** table to create your signals. The number of block inputs depends on the number of signals you specify.

Function Block Parameters: CAN Pack

Pack data into a CAN Message.

Parameters

Data is input as: manually specified signals

CANdb file: Browse...

Message list: (none)

Message

Name: CAN Msg

Identifier type: Standard (11-bit identifier)

Identifier: 250

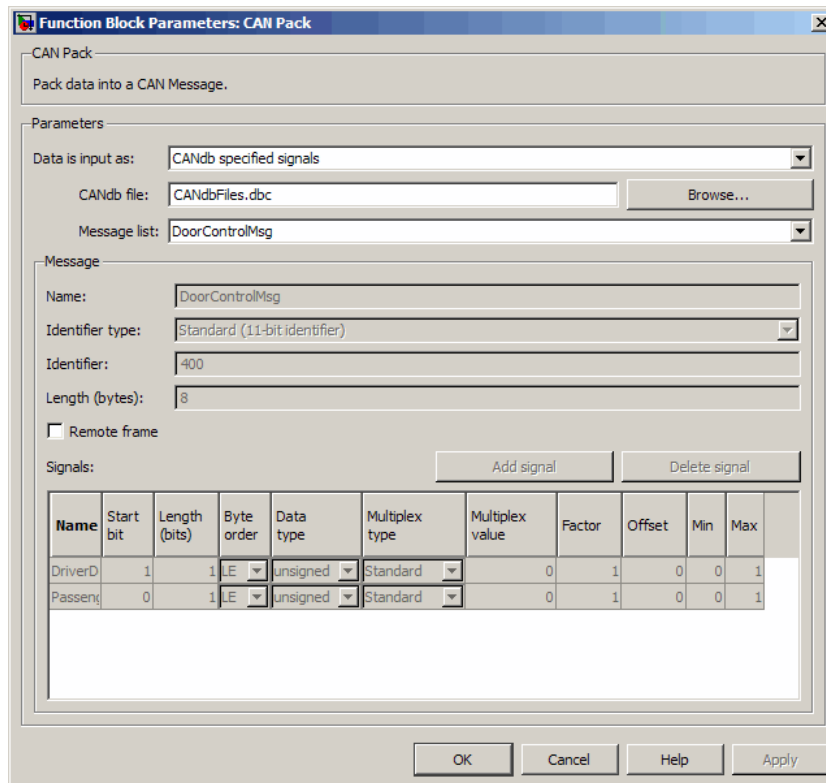
Length (bytes): 8

Remote frame

Signals:

Name	Start bit	Length (bits)	Byte order	Data type	Multiplex type	Multiplex value	Factor	Offset	Min	Max
Signal1	0	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal2	8	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal3	16	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal4	24	8	LE	signed	Standard	0	1	0	-Inf	Inf

- **CANdb specified signals:** Allows you to specify a CAN database file that contains message and signal definitions. If you select this option, select a CANdb file. The number of block inputs depends on the number of signals specified in the CANdb file for the selected message.



CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** list. Click **Browse** to find the CANdb file on your system. The message list specified in the CANdb file populates the **Message** section of the dialog box. The CANdb file also populates the **Signals** table for the selected message.

Note: File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

Message list

This option is available if you specify that your data is input via a CANdb file in the **Data is input as** field and you select a CANdb file in the **CANdb file** field. Select the message to display signal details in the **Signals** table.

Message

Name

Specify a name for your CAN message. The default is **CAN Msg**. This option is available if you choose to input raw data or manually specify signals. This option is unavailable if you choose to use signals from a CANdb file.

Identifier type

Specify whether your CAN message identifier is a **Standard** or an **Extended** type. The default is **Standard**. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to input raw data or manually specify signals. For **CANdb specified signals**, the **Identifier type** inherits the type from the database.

Identifier

Specify your CAN message ID. This number must be a positive integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. You can also specify hexadecimal values using the **hex2dec** function. This option is available if you choose to input raw data or manually specify signals.

Length (bytes)

Specify the length of your CAN message from 0 to 8 bytes. If you are using **CANdb specified signals** for your data input, the CANdb file defines the length of your message. If not, this field defaults to **8**. This option is available if you choose to input raw data or manually specify signals.

Remote frame

Specify the CAN message as a remote frame.

Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is **Signal [row number]**.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message data. The start bit must be an integer from 0 through 63.

Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

Select either of the following options:

- **LE:** Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.

Bit Number		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	Byte 0	7	6	5	4	3	2	1	0
	Byte 1	15	14	13	12	11	10	9	8
	Byte 2	23	22	21	20	19	18	17	16
	Byte 3	31	30	29	28	27	26	25	24
	Byte 4	39	38	37	36	35	34	33	32
	Byte 5	47	46	45	44	43	42	41	40
	Byte 6	55	54	53	52	51	50	49	48
	Byte 7	63	62	61	60	59	58	57	56

Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where byte order is in big-endian format (Motorola®). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.

		Bit Number							
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	Byte 0	7	6	5	4	3	2	1	0
	Byte 1	15	14	13	12	11	10	9	8
	Byte 2	23	22	21	20	19	18	17	16
	Byte 3	31	30	29	28	27	26	25	24
	Byte 4	39	38	37	36	35	34	33	32
	Byte 5	47	46	45	44	43	42	41	40
	Byte 6	55	54	53	52	51	50	49	48
	Byte 7	63	62	61	60	59	58	57	56

Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- signed (default)
- unsigned
- single
- double

Multiplex type

Specify how the block packs the signals into the CAN message at each timestep:

- **Standard:** The signal is packed at each timestep.
- **Multiplexor:** The **Multiplexor** signal, or the mode signal is packed. You can specify only one **Multiplexor** signal per message.
- **Multiplexed:** The signal is packed if the value of the **Multiplexor** signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following types and values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	N/A
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	N/A

In this example:

- The block packs Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block packs Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block packs Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not pack either of the Multiplexed signals in that timestep.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be **Multiplexed**. The value you provide here must match the **Multiplexor** signal value at run time for the block to pack the **Multiplexed** signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 2-30 to understand how physical values are converted to raw values packed into a message.

Offset

Specify the **Offset** value to apply to convert the physical value (signal value) to the raw value packed in the message. See “Conversion Formula” on page 2-30 to understand how physical values are converted to raw values packed into a message.

Min

Specify the minimum physical value of the signal. The default value is `-inf` (negative infinity). You can specify a number for the minimum value. See “Conversion Formula” on page 2-30 to understand how physical values are converted to raw values packed into a message.

Max

Specify the maximum physical value of the signal. The default value is `inf`. You can specify a number for the maximum value. See “Conversion Formula” on page 2-30 to understand how physical values are converted to raw values packed into a message.

Conversion Formula

The conversion formula is

$$\text{raw_value} = (\text{physical_value} - \text{Offset}) / \text{Factor}$$

where `physical_value` is the value of the signal after it is saturated using the specified **Min** and **Max** values. `raw_value` is the packed signal value.

See Also

CAN Unpack

CAN Unpack

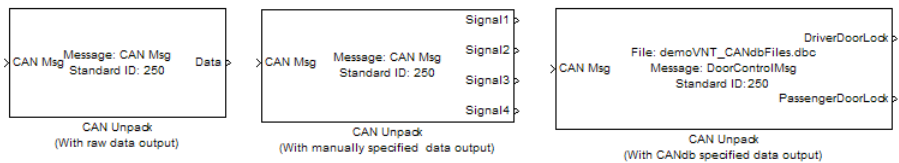
Unpack individual signals from CAN messages

Library

CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

Description



The CAN Unpack block unpacks a CAN message into signal data using the specified output parameters at every timestep. Data is output as individual signals.

Note: To use this block, you also need a license for Simulink software.

The CAN Unpack block has one output port by default. The number of output ports is dynamic and depends on the number of signals you specify for the block to output. For example, if your block has four signals, it has four output ports.

Other Supported Features

The CAN Unpack block supports:

- The use of Simulink Accelerator Rapid Accelerator mode. Using this feature, you can speed up the execution of Simulink models.

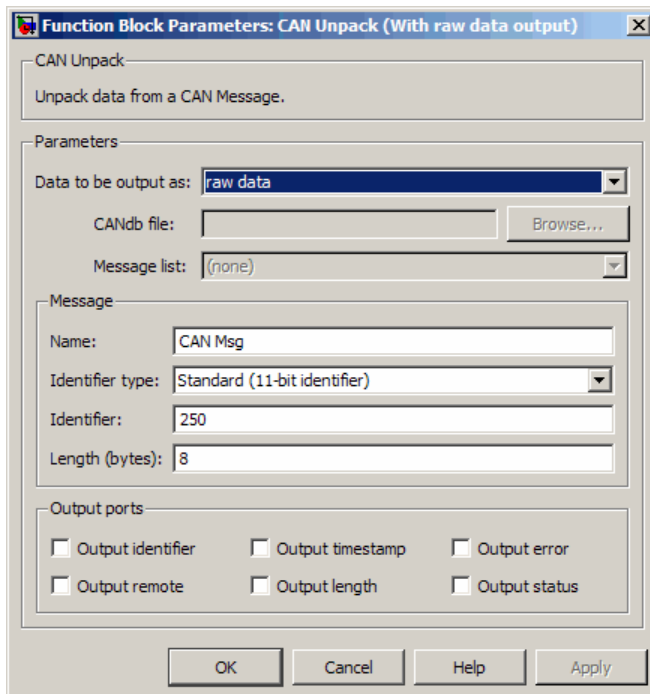
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Simulink Coder to deploy models to targets.

Note: Code generation is not supported if your signal information consists of signed or unsigned integers greater than 32-bits long.

For more information on these features, see the Simulink documentation.

Dialog Box

Use the Function Block Parameters dialog box to select your CAN message unpacking parameters.



Parameters

Data to be output as

Select your data signal:

- **raw data:** Output data as a uint8 vector array. If you select this option, you only specify the message fields. The other signal parameter fields are unavailable. This option opens only one output port on your block.
- **manually specified signals:** Allows you to specify data signals. If you select this option, use the **Signals** table to create your signals message manually.

Function Block Parameters: CAN Unpack (With manually specified data output)

—CAN Unpack—
Unpack data from a CAN Message.

Parameters

Data to be output as: manually specified signals

CANdb file: Browse...

Message list: (none)

Message

Name: CAN Msg

Identifier type: Standard (11-bit identifier)

Identifier: 250

Length (bytes): 8

Signals:

Name	Start bit	Length (bits)	Byte order	Data type	Multiplex type	Multiplex value	Factor	Offset	Min	Max
Signal1	0	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal2	8	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal3	16	8	LE	signed	Standard	0	1	0	-Inf	Inf
Signal4	24	8	LE	signed	Standard	0	1	0	-Inf	Inf

Output ports

Output identifier Output timestamp Output error

Output remote Output length Output status

The number of output ports on your block depends on the number of signals you specify. For example, if you specify four signals, your block has four output ports.

- **CANdb specified signals:** Allows you to specify a CAN database file that contains data signals. If you select this option, select a CANdb file.

Function Block Parameters: CAN Unpack (With CANdb specified data output)

—CAN Unpack—
Unpack data from a CAN Message.

Parameters

Data to be output as: CANdb specified signals

CANdb file: CANdbFiles.dbc Browse...

Message list: DoorControlMsg

Message

Name: DoorControlMsg

Identifier type: Standard (11-bit identifier)

Identifier: -400

Length (bytes): 8

Signals: Add signal Delete signal

Name	Start bit	Length (bits)	Byte order	Data type	Multiplex type	Multiplex value	Factor	Offset	Min	Max
DriverD	1	1	LE	unsigned	Standard	0	1	0	0	1
Passen	0	1	LE	unsigned	Standard	0	1	0	0	1

Output ports

Output identifier Output timestamp Output error

Output remote Output length Output status

OK Cancel Help Apply

The number of output ports on your block depends on the number of signals specified in the CANdb file. For example, if the selected message in the CANdb file has four signals, your block has four output ports.

CANdb file

This option is available if you specify that your data is input via a CANdb file in the **Data to be output as** list. Click **Browse** to find the CANdb file on your system. The messages and signal definitions specified in the CANdb file populate the **Message** section of the dialog box. The signals specified in the CANdb file populate **Signals** table.

Note: File names that contain non-alphanumeric characters such as equal signs, ampersands, and so forth are not valid CAN database file names. You can use periods in your database name. Rename CAN database files with non-alphanumeric characters before you use them.

Message list

This option is available if you specify that your data is to be output as a CANdb file in the **Data to be output as** list and you select a CANdb file in the **CANdb file** field. You can select the message that you want to view. The **Signals** table then displays the details of the selected message.

Message

Name

Specify a name for your CAN message. The default is **CAN Msg**. This option is available if you choose to output raw data or manually specify signals.

Identifier type

Specify whether your CAN message identifier is a **Standard** or an **Extended** type. The default is **Standard**. A standard identifier is an 11-bit identifier and an extended identifier is a 29-bit identifier. This option is available if you choose to output raw data or manually specify signals. For CANdb-specified signals, the **Identifier type** inherits the type from the database.

Identifier

Specify your CAN message ID. This number must be a integer from 0 through 2047 for a standard identifier and from 0 through 536870911 for an extended identifier. If you specify **-1**, the block unpacks the messages that match the length specified for the message. You can also specify hexadecimal values using the **hex2dec** function. This option is available if you choose to output raw data or manually specify signals.

Length (bytes)

Specify the length of your CAN message from 0 to 8 bytes. If you are using CANdb specified signals for your output data, the CANdb file defines the length of your message. If not, this field defaults to 8. This option is available if you choose to output raw data or manually specify signals.

Signals Table

This table appears if you choose to specify signals manually or define signals using a CANdb file.

If you are using a CANdb file, the data in the file populates this table automatically and you cannot edit the fields. To edit signal information, switch to manually specified signals.

If you have selected to specify signals manually, create your signals manually in this table. Each signal you create has the following values:

Name

Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is **Signal [row number]**.

Start bit

Specify the start bit of the data. The start bit is the least significant bit counted from the start of the message. The start bit must be an integer from 0 through 63.

Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

Select either of the following options:

- **LE:** Where the byte order is in little-endian format (Intel). In this format you count bits from the start, which is the least significant bit, to the most significant bit, which has the highest bit index. For example, if you pack one byte of data in little-endian format, with the start bit at 20, the data bit table resembles this figure.

		Bit Number							
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number	Byte 0	7	6	5	4	3	2	1	0
	Byte 1	15	14	13	12	11	10	9	8
	Byte 2	23	22	21	20	19	18	17	16
	Byte 3	31	30	29	28	27	26	25	24
	Byte 4	39	38	37	36	35	34	33	32
	Byte 5	47	46	45	44	43	42	41	40
	Byte 6	55	54	53	52	51	50	49	48
	Byte 7	63	62	61	60	59	58	57	56

Little-Endian Byte Order Counted from the Least Significant Bit to the Highest Address

- BE: Where the byte order is in big-endian format (Motorola). In this format you count bits from the start, which is the least significant bit, to the most significant bit. For example, if you pack one byte of data in big-endian format, with the start bit at 20, the data bit table resembles this figure.

Bit Number		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Data Byte Number		7	6	5	4	3	2	1	0
	Byte 0								
	Byte 1	15	14	13	12	11	10	9	8
						MSB			
	Byte 2	23	22	21	20	19	18	17	16
					LSB				
	Byte 3	31	30	29	28	27	26	25	24
Byte 4	39	38	37	36	35	34	33	32	
Byte 5	47	46	45	44	43	42	41	40	
Byte 6	55	54	53	52	51	50	49	48	
Byte 7	63	62	61	60	59	58	57	56	

Big-Endian Byte Order Counted from the Least Significant Bit to the Lowest Address

Data type

Specify how the signal interprets the data in the allocated bits. Choose from:

- signed (default)
- unsigned
- single
- double

Multiplex type

Specify how the block unpacks the signals from the CAN message at each timestep:

- **Standard:** The signal is unpacked at each timestep.
- **Multiplexor:** The **Multiplexor** signal, or the mode signal is unpacked. You can specify only one **Multiplexor** signal per message.
- **Multiplexed:** The signal is unpacked if the value of the **Multiplexor** signal (mode signal) at run time matches the configured **Multiplex value** of this signal.

For example, a message has four signals with the following values.

Signal Name	Multiplex Type	Multiplex Value
Signal-A	Standard	N/A
Signal-B	Multiplexed	1
Signal-C	Multiplexed	0
Signal-D	Multiplexor	N/A

In this example:

- The block unpacks Signal-A (Standard signal) and Signal-D (Multiplexor signal) in every timestep.
- If the value of Signal-D is 1 at a particular timestep, then the block unpacks Signal-B along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is 0 at a particular timestep, then the block unpacks Signal-C along with Signal-A and Signal-D in that timestep.
- If the value of Signal-D is not 1 or 0, the block does not unpack either of the Multiplexed signals in that timestep.

Multiplex value

This option is available only if you have selected the **Multiplex type** to be **Multiplexed**. The value you provide here must match the **Multiplexor** signal value at run time for the block to unpack the **Multiplexed** signal. The **Multiplex value** must be a positive integer or zero.

Factor

Specify the **Factor** value applied to convert the unpacked raw value to the physical value (signal value). See “Conversion Formula” on page 2-41 to understand how unpacked raw values are converted to physical values.

Offset

Specify the **Offset** value applied to convert the physical value (signal value) to the unpacked raw value. See “Conversion Formula” on page 2-41 to understand how unpacked raw values are converted to physical values.

Min

Specify the minimum raw value of the signal. The default value is `-inf` (negative infinity). You can specify a number for the minimum value. See “Conversion Formula” on page 2-41 to understand how unpacked raw values are converted to physical values.

Max

Specify the maximum raw value of the signal. The default value is `inf`. You can specify a number for the maximum value. See “Conversion Formula” on page 2-41 to understand how unpacked raw values are converted to physical values.

Output Ports

Selecting an **Output ports** option adds an output port to your block.

Output identifier

Select this option to output a CAN message identifier. The data type of this port is **uint32**.

Output remote

Select this option to output the message remote frame status. This option adds a new output port to the block. The data type of this port is **uint8**.

Output timestamp

Select this option to output the message time stamp. This option adds a new output port to the block. The data type of this port is **double**.

Output length

Select this option to output the length of the message in bytes. This option adds a new output port to the block. The data type of this port is **uint8**.

Output error

Select this option to output the message error status. This option adds a new output port to the block. The data type of this port is **uint8**.

Output status

Select this option to output the message received status. The status is `1` if the block receives new message and `0` if it does not. This option adds a new output port to the block. The data type of this port is **uint8**.

If you do not select an **Output ports** option, the number of output ports on your block depends on the number of signals you specify.

Conversion Formula

The conversion formula is

$$\text{physical_value} = \text{raw_value} * \text{Factor} + \text{Offset}$$

where **raw_value** is the unpacked signal value. **physical_value** is the scaled signal value which is saturated using the specified **Min** and **Max** values.

See Also

CAN Pack

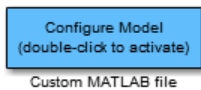
Custom MATLAB file

Automatically update active configuration parameters of parent model using file containing custom MATLAB code

Library

Configuration Wizards

Description



When you add a Custom MATLAB file block to your Simulink model and double-click it, a custom MATLAB script executes and automatically configures model parameters that are relevant to code generation. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

MathWorks provides an example MATLAB script, `matlabroot/toolbox/rtw/rtw/rtwsampleconfig.m`, that you can use with the Custom MATLAB file block and adapt to your model requirements. The block and the script provide a starting point for customization. For more information, see “Create a Custom Configuration Wizard Block” in the Embedded Coder documentation.

Note: You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- ERT (optimized for fixed-point)
- ERT (optimized for floating-point)
- GRT (optimized for fixed/floating-point)
- GRT (debug for fixed/floating-point)
- Custom

For this block, `Custom` is selected by default.

Configuration function

Name of the predefined or custom MATLAB script to be used to update the active configuration parameters of the parent Simulink model. The default value is `rtwsampleconfig`, which refers to the example script `rtwsampleconfig.m`.

Invoke build process after configuration

If selected, the script initiates the code generation and build process after updating the model's configuration parameters. If not selected (the default), the build process is not initiated.

See Also

ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

“Wizard” in the Embedded Coder documentation

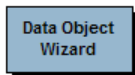
Data Object Wizard

Simulink data object wizard for creating potential Simulink data objects

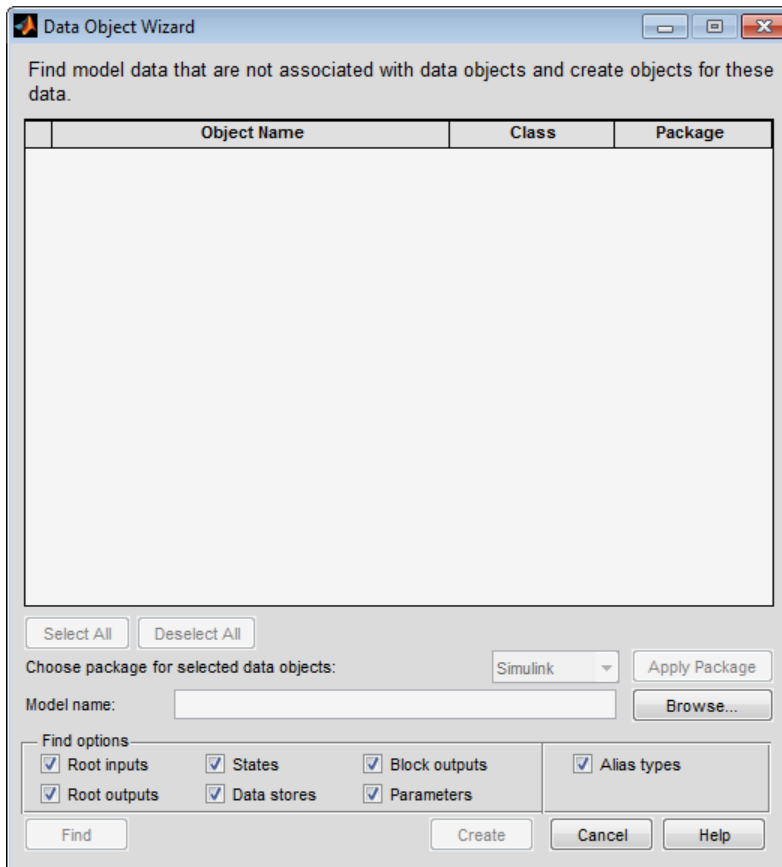
Library

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.

You can also launch the Data Object Wizard by entering `dataobjectwizard` at the MATLAB command line or by selecting **Code > Data Objects > Data Object Wizard** in your model.

Example

For an example of a model that incorporates the Data Object Wizard block, see `rtwdemo_mpf`.

See Also

“Create Data Objects with Data Object Wizard”

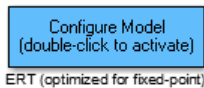
ERT (optimized for fixed-point)

Automatically update active configuration parameters of parent model for ERT fixed-point code generation

Library

Configuration Wizards

Description



When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note: You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- ERT (optimized for fixed-point)
- ERT (optimized for floating-point)

- GRT (optimized for fixed/floating-point)
- GRT (debug for fixed/floating-point)
- Custom

For this block, ERT (optimized for fixed-point) is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the “Custom MATLAB 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 MATLAB file”, ERT (optimized for floating-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

“Wizard” in the Embedded Coder documentation

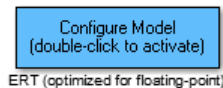
ERT (optimized for floating-point)

Automatically update active configuration parameters of parent model for ERT floating-point code generation

Library

Configuration Wizards

Description



When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for floating-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note: You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- ERT (optimized for fixed-point)
- ERT (optimized for floating-point)
- GRT (optimized for fixed/floating-point)

- GRT (debug for fixed/floating-point)
- Custom

For this block, ERT (optimized for floating-point) is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to **Custom**. This parameter is used with the “Custom MATLAB 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 MATLAB file”, ERT (optimized for fixed-point), GRT (debug for fixed/floating-point), GRT (optimized for fixed/floating-point)

“Wizard” in the Embedded Coder documentation

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

Library

Configuration Wizards

Description



GRT (debug for fixed/floating-point)

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed/floating-point code generation, with TLC debugging options enabled, with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note: You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- ERT (optimized for fixed-point)
- ERT (optimized for floating-point)

- GRT (optimized for fixed/floating-point)
- GRT (debug for fixed/floating-point)
- Custom

For this block, GRT (debug for fixed/floating-point) is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to **Custom**. This parameter is used with the “Custom MATLAB 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 MATLAB file”, ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (optimized for fixed/floating-point)

“Wizard” in the Embedded Coder documentation

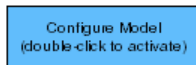
GRT (optimized for fixed/floating-point)

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Library

Configuration Wizards

Description



GRT (optimized for fixed/floating-point)

When you add a GRT (optimized for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed/floating-point code generation with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note: You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- ERT (optimized for fixed-point)
- ERT (optimized for floating-point)

- GRT (optimized for fixed/floating-point)
- GRT (debug for fixed/floating-point)
- Custom

For this block, GRT (optimized for fixed/floating-point) is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the “Custom MATLAB 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 MATLAB file”, ERT (optimized for fixed-point), ERT (optimized for floating-point), GRT (debug for fixed/floating-point)

“Wizard” in the Embedded Coder documentation

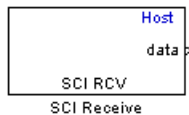
Host SCI Receive

Configure host-side serial communications interface to receive data from serial port

Library

Embedded Coder/ Embedded Targets/ Host Communication

Description



Specify the configuration of data being received from the target by this block.

The data package being received is limited to 16 bytes of ASCII characters, including package headers and terminators. Calculate the size of a package by including the package header, or terminator, or both, and the data size.

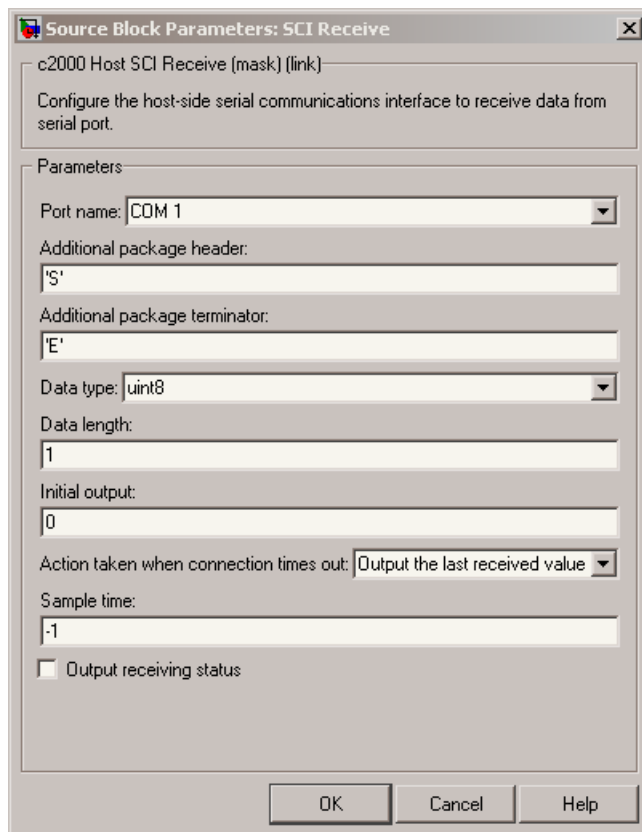
Acceptable data types are single, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The number of bytes in each data type is listed in the following table:

Data Type	Byte Count
single	4 bytes
<code>int8</code> and <code>uint8</code>	1 byte
<code>int16</code> and <code>uint16</code>	2 bytes
<code>int32</code> and <code>uint32</code>	4 bytes

For example, if your data package has package header 'S' (1 byte) and package terminator 'E' (1 byte), that leaves 14 bytes for the actual data. If your data is of type `int8`, there is room in the data package for 14 `int8`s. If your data is of type `uint16`, there is room in the data package for 7 `uint16`s. If your data is of type `int32`, there is room in the data package for only 3 `int32`s, with 2 bytes left over. Even though you could fit two `int8`s or one `uint16` in the remaining space, you may not, because you cannot mix data types in the same package.

The number of data types that can fit into a data package determine the data length (see **Data length** in the Dialog Box description). In the example just given, the 14 for data type `int8` and the 7 for data type `uint16` are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run time errors, may result.

Dialog Box



Port name

You may configure up to four COM ports (COM1 through COM4) for up to four host-side SCI Receive blocks.

Additional package header

This field specifies the data located at the front of the received data package, which is not part of the data being received, and generally indicates start of data. The additional package header must be an ASCII value. You can use a string or number (0–255). You must put single quotes around strings entered in this field, but the quotes are not received nor are they included in the total byte count.

Note: Match additional package headers or terminators with those specified in the target SCI transmit block.

Additional package terminator

This field specifies the data located at the end of the received data package, which is not part of the data being received, and generally indicates end of data. The additional package terminator must be an ASCII value. You can use a string or number (0–255). You must put single quotes around strings entered in this field, but the quotes are not received nor are they included in the total byte count.

Data type

Choice of single, int8, uint8, int16, uint16, int32, or uint32.

The input port of the SCI Transmit block accepts only one of these values. Which value it accepts is inherited from the data type from the input (the data length is also inherited from the input). Data must consist of only one data type; you cannot mix types.

Data length

How many of **Data type** the block receives (not bytes). Anything more than 1 is a vector. The data length is inherited from the input (the data length input to the SCI Transmit block).

Initial output

Default value from the SCI Receive block. This value is used, for example, if a connection time-out occurs and the **Action taken when connection timeout** field is set to “Output the last received value”, but nothing yet has been received.

Action Taken when connection times out

Specify what to output if a connection time-out occurs. If “Output the last received value” is selected, the block outputs the last received value. If a value has not been received, the block outputs the **Initial output**.

If you select **Output custom value**, use the **Output value when connection times out** field to set the custom value.

Sample time

Determines how often the SCI Receive block is called (in seconds). When you set this value to -1, the model inherits the sample time value of the model. To execute this block asynchronously, set **Sample Time** to -1, and refer to “” for a discussion of block placement and other settings.

Output receiving status

Selecting this checkbox creates a **Status** block output that provides the status of the transaction.

The error status may be one of the following values:

- 0: No errors
- 1: A time-out occurred while the block was waiting to receive data
- 2: There is an error in the received data (checksum error)
- 3: SCI parity error flag — Occurs when a character is received with a mismatch
- 4: SCI framing error flag — Occurs when an expected stop bit is not found

See Also

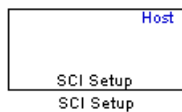
Host SCI Setup

Configure COM ports for host-side SCI Transmit and Receive blocks

Library

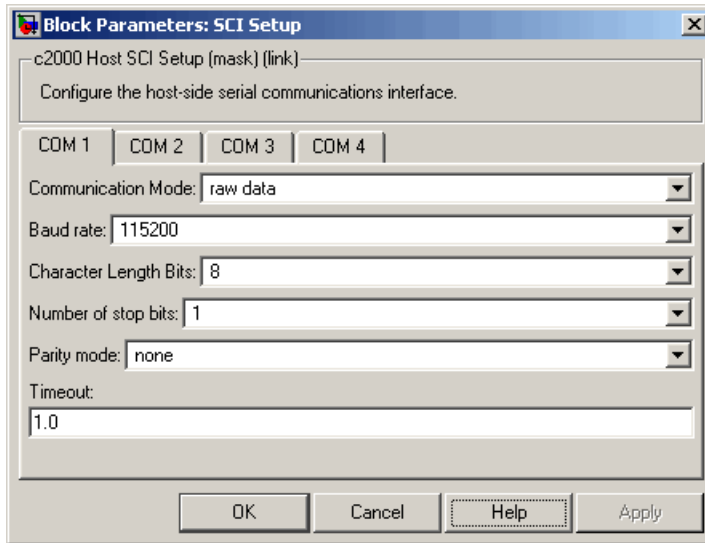
Embedded Coder/ Embedded Targets/ Host Communication

Description



Standardize COM port settings for use by the host-side SCI Transmit and Receive blocks. Setting COM port configurations globally with the SCI Setup block avoids conflicts (e.g., the host-side SCI Transmit block cannot use COM1 with settings different than those the COM1 used by the host-side SCI Receive block) and requires that you set configurations only once for each COM port. The SCI Setup block is a stand alone block.

Dialog Box



Communication Mode

Raw data or protocol. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlocks do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

If you specify protocol mode, some handshaking between host and target occurs. The transmitting side sends \$SND indicating that it is ready to transmit. The receiving side sends back \$RDY indicating that it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include

- Data is received as expected (checksum)
- Data is received by target
- Time consistency; each side waits for its turn to send or receive

Note: Deadlocks can occur if one SCI Transmit block is trying to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

Baud rate

Choose from 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

Number of stop bits

Select 1 or 2.

Parity mode

Select none, odd, or even.

Timeout

Enter values greater than or equal to 0, in seconds. When the COM port involved is using protocol mode, this value indicates how long the transmitting side waits for an acknowledgement from the receiving side or how long the receiving side waits for data. The system displays a warning message if the time-out is exceeded, every n number of seconds, n being the value in **Timeout**.

Note: Simulink suspends processing for the length of the time-out. During that time you cannot perform actions in Simulink. If the time-out is set for a long period of time, it may appear that Simulink has frozen.

See Also

Host SCI Transmit

Configure host-side serial communications interface to transmit data to serial port

Library

Embedded Coder/ Embedded Targets/ Host Communication

Description



Specify the configuration of data being transmitted to the target from this block.

The data package being sent is limited to 16 bytes of ASCII characters, including package headers and terminators. Calculate the size of a package by figuring in package header, or terminator, or both, and the data size.

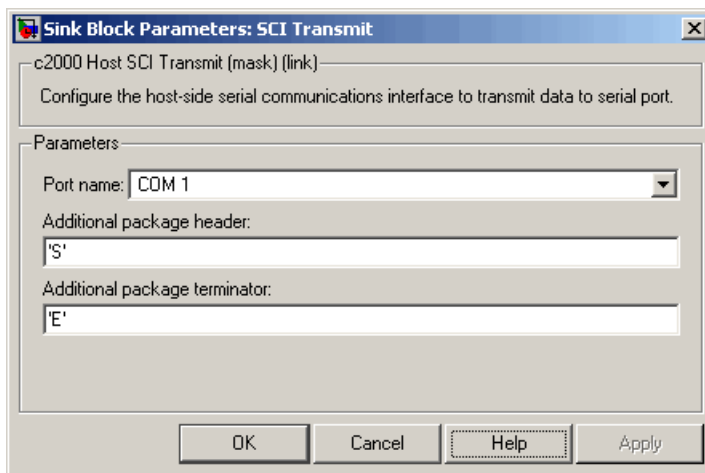
Acceptable data types are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The byte size of each data type is as follows:

Data Type	Byte Count
<code>single</code>	4 bytes
<code>int8</code> & <code>uint8</code>	1 byte
<code>int16</code> & <code>uint16</code>	2 bytes
<code>int32</code> & <code>uint32</code>	4 bytes

For example, if your data package has package header “S” (1 byte) and package terminator “E” (1 byte), that leaves 14 bytes for the actual data. If your data is of type `int8`, there is room in the data package for 14 `int8`s. If your data is of type `uint16`, there is room in the data package for only 7 `uint16`s. If your data is of type `int32`, there is room in the data package for only 3 `int32`s, with 2 bytes left over. Even though you could fit two `int8`s or one `uint16` in the remaining space, you may not, because you cannot mix data types in the same package.

The number of data types that can fit into a data package determine the data length (see **Data length** in the Dialog Box description). In the example just given, the 14 for data type int8 and the 7 for data type uint16 are the data lengths for each data package, respectively. When the data length exceeds 16 bytes, unexpected behavior, including run time errors, may result.

Dialog Box



Port name

You may configure up to four COM ports (COM1 through COM4) for up to four host-side SCI Transmit blocks.

Additional package header

This field specifies the data located at the front of the transmitted data package, which is not part of the data being transmitted, and generally indicates start of data. The additional package header must be an ASCII value. You can use a string or number (0–255). You must put single quotes around strings entered in this field, but the quotes are not sent nor are they included in the total byte count.

Note: Match additional package headers or terminators with those specified in the target SCI receive block.

Additional package terminator

This field specifies the data located at the end of the transmitted data package, which is not part of the data being sent, and generally indicates end of data. The additional package terminator must be an ASCII value. You can use a string or number (0–255). You must put single quotes around strings entered in this field, but the quotes are not transmitted nor are they included in the total byte count.

See Also

Idle Task

Create free-running task

Description



The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. The tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

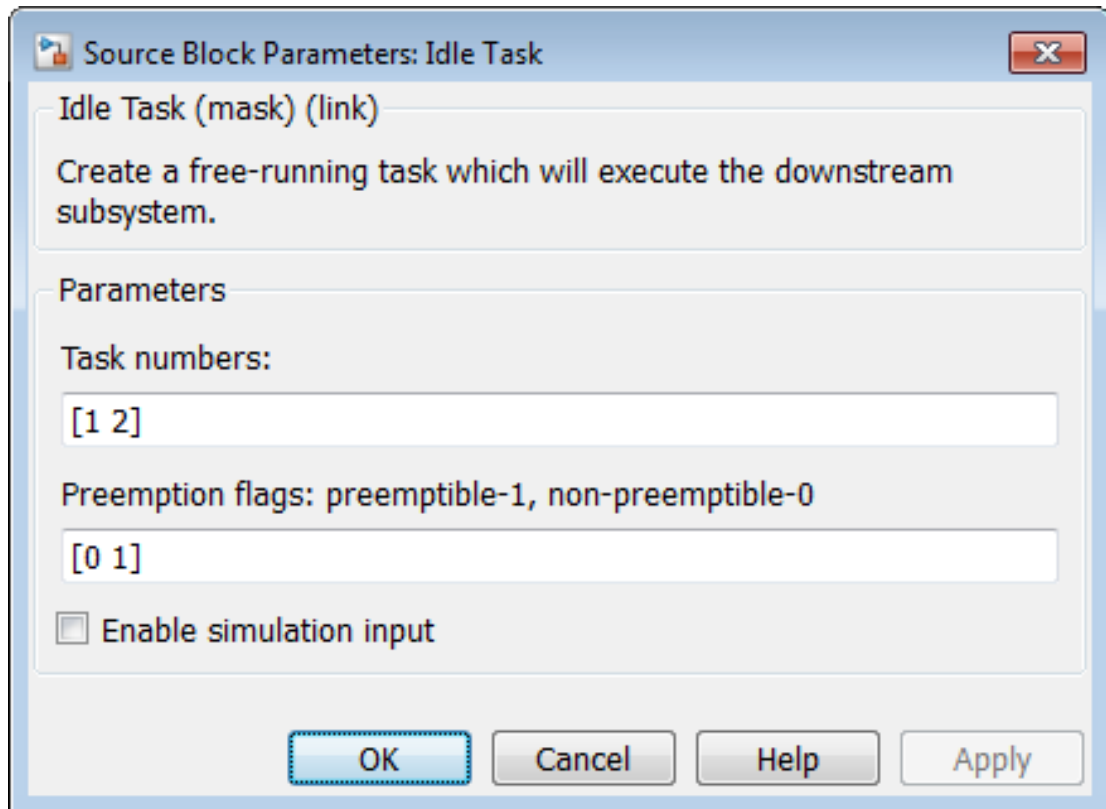
This block is not supported on targets running an operating system or RTOS.

Vectorized Output

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. A preemption-flag vector must be the same length as the number of tasks vector unless the preemption flag vector has only one element. The value of the preemption flag determines whether a given interrupt (and task) is preemptible. Preemption overrides prioritization. A lower-priority nonpreemptible task can preempt a higher-priority preemptible task.

When the preemption flag vector has one element, that element value applies to the functions in the downstream subsystem as defined by the task numbers in the task number vector. If the preemption flag vector has the same number of elements as the task number vector, each task defined in the task number vector has a preemption status defined by the value of the corresponding element in the preemption flag vector.

Dialog Box



Task numbers

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers. The default values are [1 , 2] to indicate that the downstream subsystem has two functions.

The values you enter determine the execution order of the functions in the downstream subsystem, while the number of values you enter corresponds to the number of functions in the downstream subsystem.

Enter a vector containing the same number of elements as the number of functions in the downstream subsystem. This vector can contain up to 16 elements, and the values must be from 0 to 15 inclusive.

The value of the first element in the vector determines the order in which the first function in the subsystem is executed, the value of the second element determines the order in which the second function in the subsystem is executed, and so on.

For example, entering [2,3,1] in this field indicates that there are three functions to be executed, and that the third function is executed first, the first function is executed second, and the second function is executed third. After the functions are executed, the Idle Task block cycles back and repeats the execution of the functions in the same order.

Preemption flags

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Task numbers** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of the tasks in **Task numbers**. If **Task numbers** contains more than one task, and you enter only one flag value here, that status applies to the tasks.

In the default settings [0 1], the task with priority 1 in **Task numbers** is not preemptible, and the priority 2 task can be preempted.

Enable simulation input

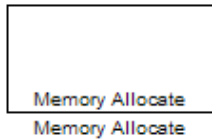
When you select this option, Simulink software adds an input port to the Idle Task block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Note Select this check box to test asynchronous interrupt processing behavior in Simulink software.

Memory Allocate

Allocate memory section

Description



On C2xxx, C5xxx, or C6xxx processors, this block directs the TI compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

The block does not verify whether the entries for your variable are valid, such as checking the variable name, data type, or section. You must check that all variable names are valid, that they use valid data types, and that all section names you specify are valid as well.

The block does not have input or output ports. It only allocates a memory location. You do not connect it to other blocks in your model.

Dialog Box

The block dialog box comprises multiple tabs:

- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

The dialog box images show all of the available parameters enabled. Some of the parameters shown do not appear until you select one or more other parameters.

Block Parameters: Memory Allocate

Memory Allocate (mask)

Allocates the target memory for the variable using specified data type and dimension. The variable may be aligned to the specified alignment boundary and may be initialized with the specified value. In addition, the variable may be placed to a specific memory section. The section may be optionally bound to a specific memory address.

Memory | Section

Variable name:
myVariable

Specify variable alignment

Memory alignment boundary:
4

Data type: uint32

Specify data type qualifier

Data type qualifier
volatile

Data dimension:
64

Initialize memory

Initial value:
0

OK Cancel Help Apply

The following sections describe the contents of each pane in the dialog box.

Memory Parameters

Block Parameters: Memory Allocate [X]

Memory Allocate (mask)

Allocates the target memory for the variable using specified data type and dimension. The variable may be aligned to the specified alignment boundary and may be initialized with the specified value. In addition, the variable may be placed to a specific memory section. The section may be optionally bound to a specific memory address.

Memory | Section

Variable name:
myVariable

Specify variable alignment

Memory alignment boundary:
4

Data type: uint32

Specify data type qualifier

Data type qualifier
volatile

Data dimension:
64

Initialize memory

Initial value:
0

OK Cancel Help Apply

You find the following memory parameters on this tab.

Variable name

Specify the name of the variable to allocate. The variable is allocated in the generated code.

Specify variable alignment

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory alignment**

boundary parameter appears so you can specify the alignment. Use this parameter and **Memory alignment boundary** when your processor requires this feature.

Memory alignment boundary

After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

Data type

Defines the data type for the variable. Select from the list of types available.

Specify data type qualifier

Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

Data type qualifier

After you select **Specify data type qualifier**, you enter the desired qualifier here. **Volatile** is the default qualifier. Enter the qualifier you need as text. Common qualifiers are **static** and **register**. The block does not check for valid qualifiers.

Data dimension

Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

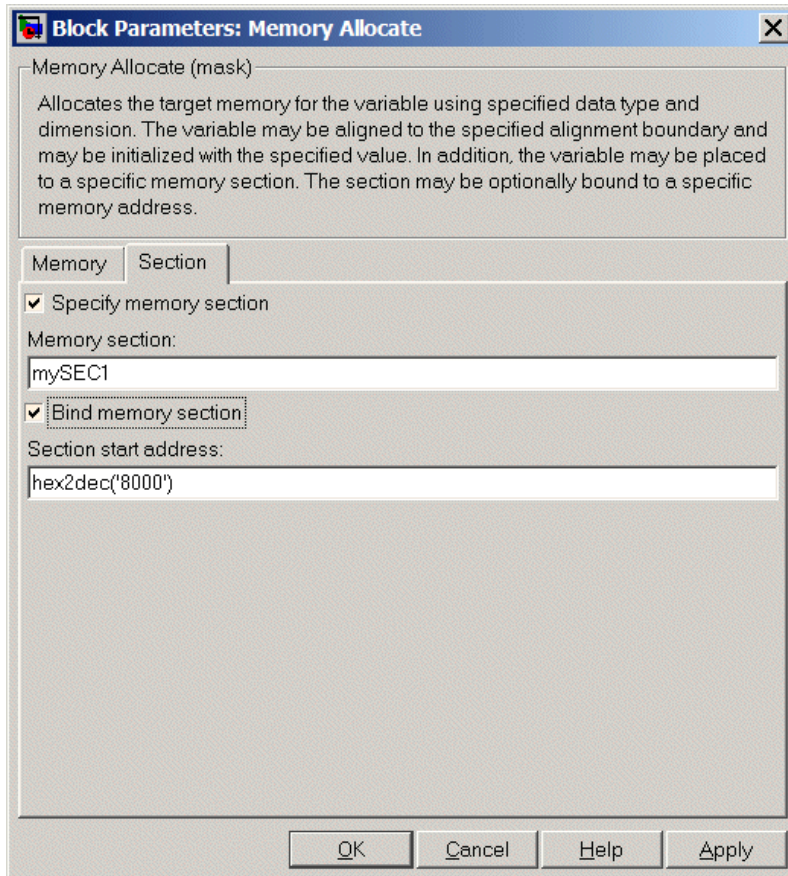
Initialize memory

Directs the block to initialize the memory location to a fixed value before processing.

Initial value

Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.

Section Parameters



Parameters on this pane specify the section in memory to store the variable.

Specify memory section

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the standard memory sections or a custom section that you declare elsewhere in your code.

Memory section

Identify a specific memory section to allocate the variable in **Variable name**. Verify that the section has enough space to store your variable. After you specify a memory

section by selecting **Specify memory section** and entering the section name in **Memory section**, use **Bind memory section** to bind the memory section to a location.

Bind memory section

After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use this parameter to bind the memory section to the location in memory specified in **Section start address**. When you select this, you enable the **Section start address** parameter.

The new memory section specified in **Memory section** is defined when you check this parameter.

Note Do not use **Bind memory section** for existing memory sections.

Section start address

Specify the address to which to bind the memory section. Enter the address in decimal form or in hexadecimal with a conversion to decimal as shown by the default value `hex2dec(' 8000 ')`. The block does not verify the address—verify that the address exists and can contain the memory section you entered in **Memory section**.

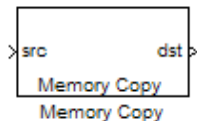
See Also

Memory Copy

Memory Copy

Copy to and from memory section

Description



In generated code, this block copies variables or data from and to processor memory as configured by the block parameters. Your model can contain as many of these blocks as you require to manipulate memory on your processor.

Each block works with one variable, address, or set of addresses provided to the block. Parameters for the block let you specify both the source and destination for the memory copy, as well as options for initializing the memory locations.

Using parameters provided by the block, you can change options like the memory stride and offset at run time. In addition, by selecting various parameters in the block, you can write to memory at program initialization, at program termination, and at every sample time. The initialization process occurs once, rather than occurring for every read and write operation.

With the custom source code options, the block enables you to add custom ANSI C source code before and after each memory read and write (copy) operation. You can use the custom code capability to lock and unlock registers before and after accessing them. For example, some processors have registers that you may need to unlock and lock with `EALLOW` and `EDIS` macros before and after your program accesses them.

If your processor or board supports quick direct memory access (QDMA) the block provides a parameter to check that implements the QDMA copy operation, and enables you to specify a function call that can indicate that the QDMA copy is finished. Only the C621x, C64xx, and C671x processor families support QDMA copy.

Note: Replace Read from Memory and Write To Memory blocks, which were removed in a previous release, with the Memory Copy block.

Block Operations

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in the three periods independently.

Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. This block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform an operation. The block output is not defined.

Copy Memory

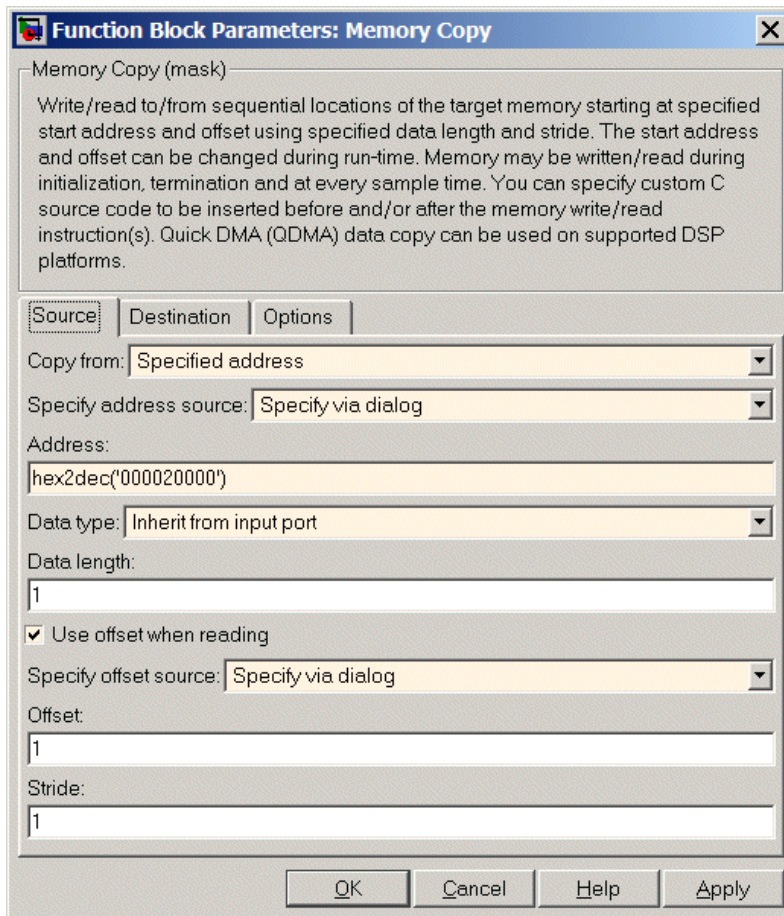
When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

Dialog Box

The block dialog box contains multiple tabs:

- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

The dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.



Sections that follow describe the parameters on each tab in the dialog box.

Source Parameters

Function Block Parameters: Memory Copy

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source | Destination | Options

Copy from: Specified address

Specify address source: Specify via dialog

Address:
hex2dec('000020000')

Data type: Inherit from input port

Data length:
1

Use offset when reading

Specify offset source: Specify via dialog

Offset:
1

Stride:
1

OK Cancel Help Apply

Copy from

Select the source of the data to copy. Choose one of the entries on the list:

- **Input port** — This source reads the data from the block input port.
- **Specified address** — This source reads the data at the specified location in **Specify address source** and **Address**.

- **Specified source code symbol** — This source tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

Note If you do not select **Input port** for **Copy from**, change **Data type** from the default **Inherit from source** to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&src**, indicating that the block expects the address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax. Enter a string to specify the symbol exactly as you use it in your code.

Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use **hex2dec** to convert the address to the expected format. The following example converts **0x1000** to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either **4096** or **hex2dec('1000')** as the address.

Data type

Use this parameter to specify the type of data that your source uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `Inherit from source` for inheriting the data type from the block input port.

Data length

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

Use offset when reading

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

Specify offset source

The block provides two sources for the offset — `Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when reading the input data.

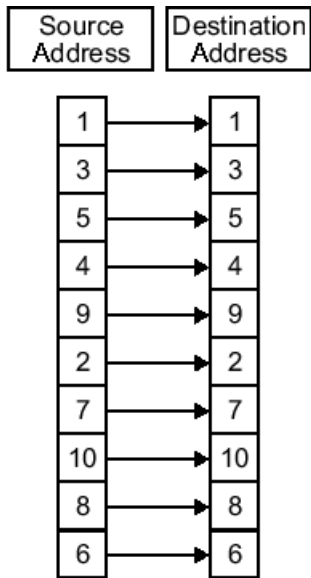
Offset

Offset tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

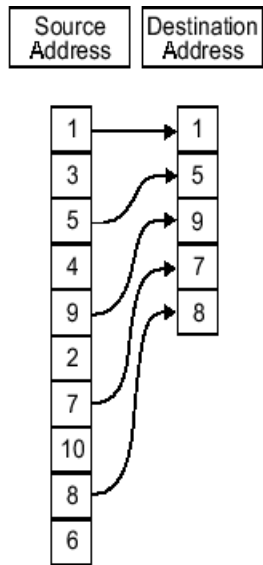
Stride

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in the source address equal to the stride. **Stride** must be a positive integer.

The next two figures help explain the stride concept. In the first figure you see data copied without a stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.

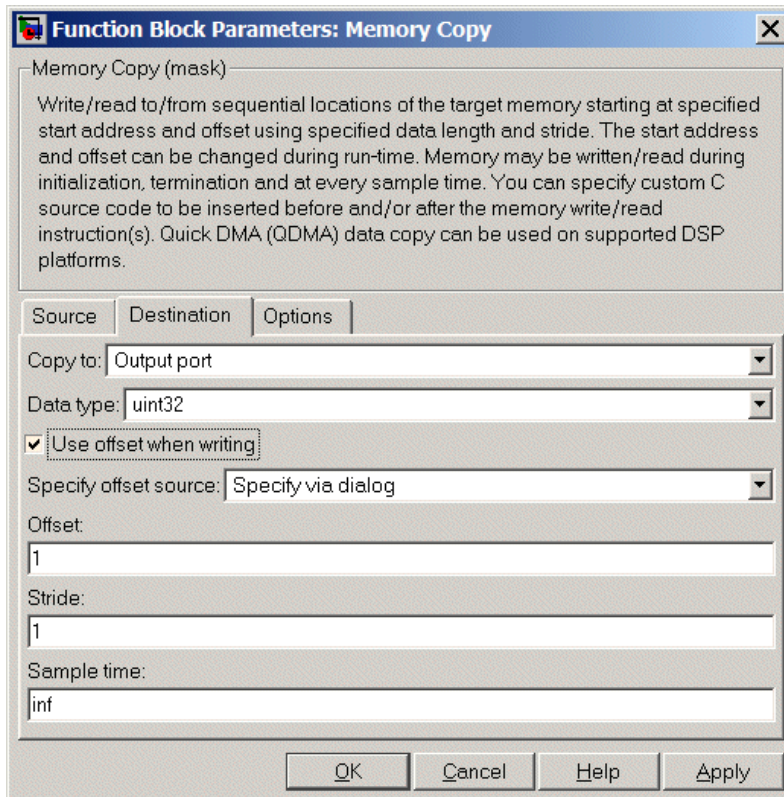


Input Stride = 1
Output Stride = 1
Number of Elements Copied = 10



Input Stride = 2
Output Stride = 1
Number of Elements Copied = 5

Destination Parameters



Copy to

Select the destination for the data. Choose one of the entries on the list:

- **Output port** — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.
- **Specified address** — Copies the data to the specified location in **Specify address source** and **Address**.
- **Specified source code symbol** — Tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

Specify address source

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&dst**, indicating that the block expects the destination address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax.

Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use **hex2dec** to convert the address to the expected format. This example converts **0x2000** to decimal form.

```
8192 = hex2dec('2000');
```

For this example, you could enter either **8192** or **hex2dec('2000')** as the address.

Data type

Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as **int8**, **uint32**, and **Boolean**, and the option **inherit from source** for inheriting the data type for the variable from the block input port.

Specify offset source

The block provides two sources for the offset—**Input port** and **Specify via dialog**. Selecting **Input port** configures the block input to read the offset value by adding an input port labeled **src ofs**. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select **Specify via dialog**, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when writing the output data.

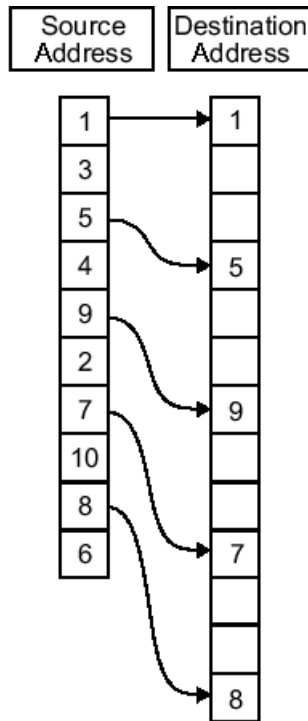
Offset

Offset tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

Stride

Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially, but by skipping addresses equal to the stride. **Stride** must be a positive integer.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.



Input Stride = 2
 Output Stride = 3
 Number of Elements Copied = 5

Sample time

Sample time sets the rate at which the memory copy operation occurs, in seconds. The default value `Inf` tells the block to use a constant sample time. You can set **Sample time** to `-1` to direct the block to inherit the sample time from the input, or from the Simulink software model when there are no block inputs. Enter the sample time in seconds as you need.

Options Parameters

Function Block Parameters: Memory Copy

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source | Destination | **Options**

Set memory value at initialization

Specify initialization value source: Specify constant value

Initialization value (constant):

1

Apply initialization value as mask

Bitwise operator: bitwise AND

Set memory value at termination

Termination value:

1

Set memory value only at initialization/termination

Insert custom code before memory write

Custom code:

/* Custom Code Before Write*/

Insert custom code after memory write

Custom code:

/* Custom Code After Write*/

Use QDMA for copy (if available)

Enable blocking mode

OK Cancel Help Apply

Set memory value at initialization

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you select this

option, use the **Set memory value at termination** and **Specify initialization value source** parameters to set your desired value. Alternately, you can tell the block to get the initial value from the block input.

Specify initialization value source

After you check **Set memory value at initialization**, use this parameter to select the source of the initial value. Choose either

- **Specify constant value** — Sets a single value to use when your program initializes memory.
- **Specify source code symbol** — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a string.

Initialization value (constant)

If you check **Set memory value at initialization** and choose **Specify constant value** for **Specify initialization value source**, enter the constant value to use in this field.

Initialization value (source code symbol)

If you check **Set memory value at initialization** and choose **Specify source code symbol** for **Specify initialization value source**, enter the symbol to use in this field. Use a valid symbol from the symbol table for the program. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

Apply initialization value as mask

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol.

Bitwise operator

To use the initialization value as a mask, select one of the entries on the following table from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

Bitwise Operator List Entry	Description
bitwise AND	Apply the mask value as a bitwise AND to the value in the register.
bitwise OR	Apply the mask value as a bitwise OR to the value in the register.
bitwise exclusive OR	Apply the mask value as a bitwise exclusive OR to the value in the register.
left shift	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
right shift	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

Set memory value at termination

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

Set memory value only at initialization/termination

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform copies during real-time operations.

Insert custom code before memory write

Select this parameter to add custom ANSI C code before the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

Custom code

Enter the custom ANSI C code to insert into the generated code just before the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Insert custom code after memory write

Select this parameter to add custom ANSI C code immediately after the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

Custom code

Enter the custom ANSI C code to insert into the generated code just after the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Use QDMA for copy (if available)

For processors that support quick direct memory access (QDMA), select this parameter to enable the QDMA operation and to access the blocking mode parameter.

If you select this parameter, your source and destination data types must be the same or the copy operation returns an error. Also, the input and output stride values must be one.

Enable blocking mode

If you select the **Use QDMA for copy** parameter, select this option to make the memory copy operations blocking processes. With blocking enabled, other processing in the program waits while the memory copy operation finishes.

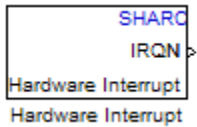
See Also

Memory Allocate

SHARC Hardware Interrupt

Generate Interrupt Service Routine

Library

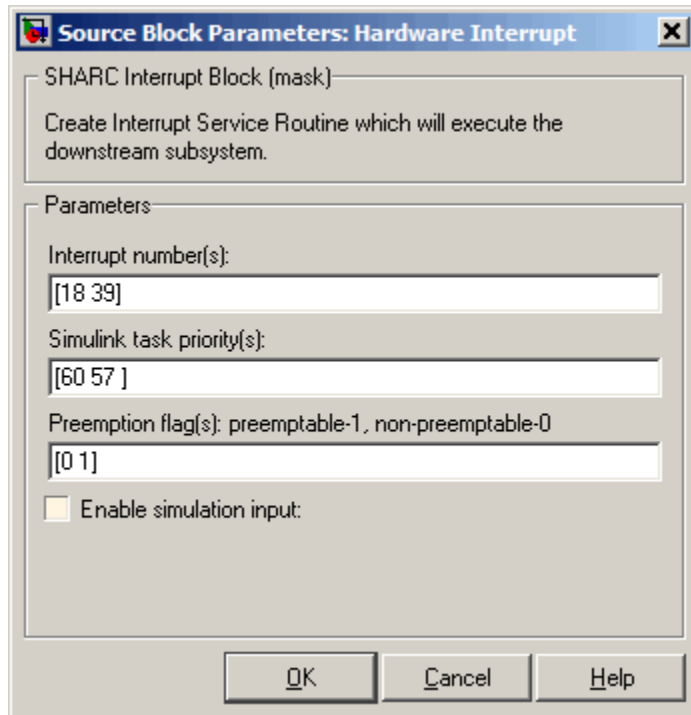


Embedded Coder/ Embedded Targets/ Processors/ Analog Devices SHARC®/ Scheduling

Description

Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that either run the processes that are downstream from this block or trigger an Idle Task block connected to this block.

Dialog Box



Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The valid ranges are 8-36 and 38-40.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. The values in this field and the **preemption flag** entries in **Preemption flags: preemptible-1, non-preemptible-0** define how the code and processor handle interrupts during asynchronous scheduler operations.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Code generation requires rate transition code (refer to “Rate Transitions and Asynchronous Blocks” in the Simulink Coder documentation). The task priority values facilitate absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags preemptible – 1, non-preemptible – 0

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering 1 indicates that the interrupt can be preempted.
- Entering 0 indicates the interrupt cannot be preempted.

When **Interrupt numbers** contains more than one interrupt value, you can assign different preemption flags to each interrupt by entering a vector of flag values to correspond to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 18 in **Interrupt numbers** is not preemptible and the priority 39 interrupt can be preempted.

Enable simulation input

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Target Preferences (Removed)

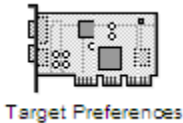
Configure model for specific IDE, tool chain, board, and processor

Library

Simulink Coder / Desktop Targets

Embedded Coder/ Embedded Targets

Description



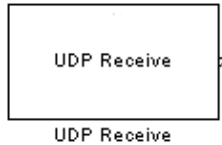
The Target Preferences block has been removed from the Simulink block libraries. The contents of the Target Preferences block have been moved to the Target Hardware Resources tab, located in the Configuration Parameters dialog. For more information, see:

- “Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box”
- “Configure Target Hardware Resources”
- “Code Generation: Coder Target Pane”

UDP Receive

Receive UDP packet

Note If your target system uses Linux[®] or Windows, get the UDP block from `linuxlib` or `windowplib`.



Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block output, emits the contents of a single UDP packet as a data vector.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Dialog

Source Block Parameters: UDP Receive

UDP Receive (mask) (link)

Receive UDP packets on a given IP port.
This block receives a UDP packet from the network and emits that data as a one-dimensional vector of the specified data type.

Parameters

Local IP port:
25000

Remote IP address ('0.0.0.0' to accept all):
'0.0.0.0'

Receive buffer size (bytes):
8192

Maximum length for Message:
255

Data type for Message: uint8

Output variable-size signal

Blocking time (seconds):
inf

Sample time (seconds):
0.01

OK Cancel Help Apply

Local IP port

Specify the IP port number upon which to receive UDP packets. This value defaults to 25000. The value can range 1–65535.

Note: On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Remote IP address ('0.0.0.0' to accept all)

Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from other addresses. To accept packets from any IP address, enter '0.0.0.0'. This value defaults to '0.0.0.0'.

Receive buffer size (bytes)

Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to 8192.

Maximum length for Message

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of a UDP packet. The system truncates data that exceeds this length. This value defaults to 255.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message

Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Output variable-size signal

If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length

Set the data type of the Length output. This option defaults to `double`.

Blocking time (seconds)

For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note: This parameter appears only in the Embedded Coder UDP Receive block.

Sample time (seconds)

Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Output port width

Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than a packet you expect to receive.

Note: This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

UDP receive buffer size (bytes)

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Note: This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

See Also

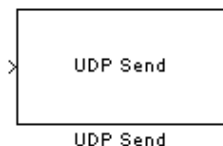
“Byte Pack”, “Byte Reversal”, “Byte Unpack”, “UDP Send”

UDP Send

Send UDP message

Note If your target system uses Linux or Windows, get the UDP block from `linuxlib` or `windowslib`.

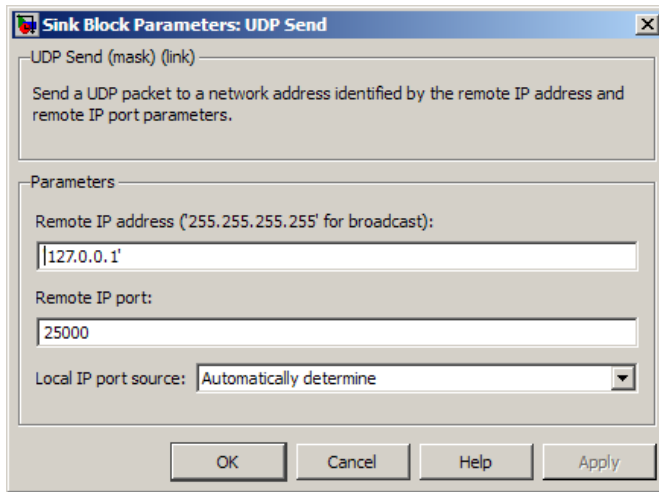
Description



The UDP Send block transmits an input vector as a UDP message over an IP network port.

Note: Some Simulink blocks and `.exe` files built from models that contain those blocks require shared libraries, such as `.dll` files on Windows. The UDP Send block requires `networkdevice.dll`. To meet this requirement, open the `packNGo` topic, and follow the example to package the code files for your model. The resulting compressed folder contains the `.dll` files that the model requires, including `networkdevice.dll`. To run this type of `.exe` file outside a MATLAB environment, place the required `.dll` files in the same folder as the `.exe` file, or place them in a folder on the Windows system path.

Dialog Box



IP address ('255.255.255.255' for broadcast)

Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, '127.0.0.1'.

Remote IP port

Specify the port to which the block sends the message. The value defaults to 25000, but the values range from 1–65535.

Note: On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Local IP port source

To let the system automatically assign the port number, select **Assign automatically**. To specify the IP port number using the **Local IP port** parameter, select **Specify**.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Sample time

Sample time tells the block how long to wait before polling for new messages.

Note: This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block.

See Also

“Byte Pack”, “Byte Reversal”, “Byte Unpack”, “UDP Receive”

C6000 IP Config

Configure Internet Protocol on C6000 targets with Ethernet ports

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ Avnet S3ADSP DM6437

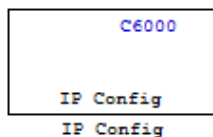
Embedded Coder Support Package for Texas Instruments C6000 Processors/ DM6437 EVM

Embedded Coder Support Package for Texas Instruments C6000 Processors/ C6747 EVM

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DM648 EVM

Embedded Coder Support Package for Texas Instruments C6000 Processors/ Target Communication

Description



Adding this block to your model provides options to configure the IP parameters for your C6000 board. Setting the options for the block sets the address and name for your board and specifies your target and Ethernet daughtercard.

To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements:

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

The block uses dynamic addressing, getting the address from the local server or static addressing. If you have a dynamic host configuration protocol (DHCP) server available, you can allow the server to provide an IP address for your board. Dynamic IP addresses can be useful but unreliable — they can change.

To use static addressing, create a static IP address by clearing **Use DHCP to allocate an IP address for DM642 EVM (requires DHCP server)**. to enable the manual IP address configuration parameters.

Note When you use the UDP Send and Receive blocks in a model, you must also include this block to set up the IP drivers for the Ethernet parameters for the target networking capability.

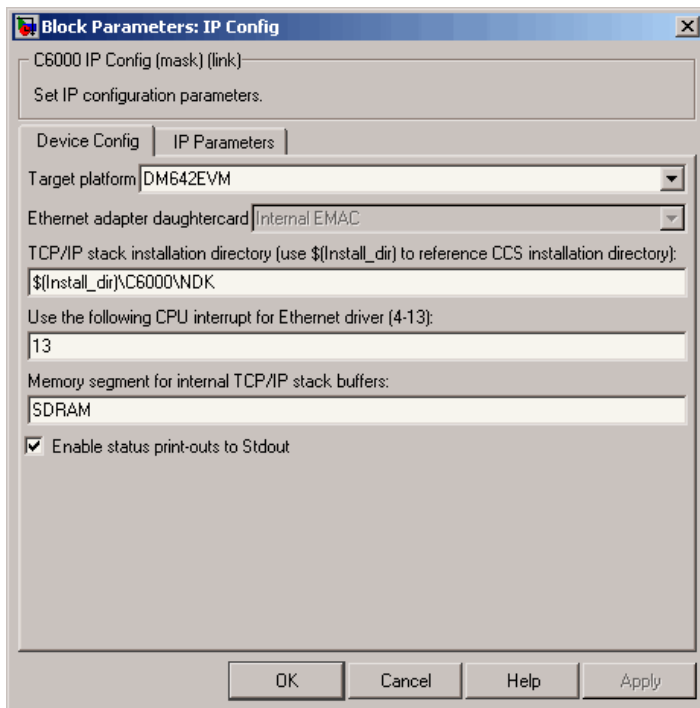
Whether you choose to use dynamic addressing, you must set the Host name, and select and set the **Use the following CPU interrupt for Ethernet driver (4-13)** options.

When you build and run your model, this block does not alter the results. It outputs zeros. When you generate code from your model, this block adds the code that configures IP on your board.

Dialog Box

The block dialog box provides options on two tabs — **Device Config** and **IP Parameters**.

Device Tab Options



Target platform

Specify your C6000 target by selecting the target board from the list. Changing the target platform changes the entry on the **Ethernet adapter daughtercard** list.

Ethernet adapter daughtercard

After you select your target platform, this option lets you select whatever daughtercard is available to implement Ethernet communications on the target.

TCP/IP stack installation folder

To use the UDP and TCP blocks for the board, you must install the TMS320C6000 TCP/IP Stack from Texas Instruments. Specify the folder where the TMS320C6000 TCP/IP Stack from Texas Instruments is installed.

Use the following CPU interrupt for Ethernet driver (4-13)

The Ethernet driver on the DM642 can respond to a CPU interrupt from 4 to 13. Enter one valid CPU interrupt for the driver to react to. CPU interrupt 13 is the default interrupt.

Memory segment for internal TCP/IP stack buffers

Shows you the segment in memory where the TCP/IP stack buffers reside. For the supported boards, the default setting and location is SDRAM. You can change the location by entering the name of the memory segment to use. TCP/IP stack buffers occupy approximately 130 kB of memory. In most cases you should locate the TCP/IP stack buffers in external memory. Be sure that the segment you specify here agrees with the memory segment allocation in the Target Hardware Resources tab.

Enable status print-outs to Stdout

Select this option to direct the block to send IP status information to the standard output device.

IP Parameters Options

Block Parameters: IP Config

C6000 IP Config (mask)
Set IP configuration parameters.

Device Config | **IP Parameters**

Use DHCP to allocate an IP address (requires a DHCP server):
Use the following IP address:
100.100.100.2

Subnet mask:
255.255.255.0

Gateway IP:
100.100.100.1

Domain name server IP:
0.0.0.0

Domain name (less than 64 characters):
mathworks.net

Host name (less than 64 characters):
dm642evm

OK Cancel Help Apply

Use DHCP to allocate an IP address (requires a DHCP server)

Selecting this parameter configures the board to get an IP address from the local DHCP server on the network. If you select this option and you do not have a DHCP server, the generated code does not run as expected. Clearing this option enables all of the IP configuration options for the block to let you define your IP address manually.

Use the following IP address

Specify an IP address. This value is the address that others use to communicate with the evaluation module over IP. Use the full xxx.xxx.xxx.xxx format.

Subnet mask

Define the subnet mask address, entering the full subnet mask in the format xxx.xxx.xxx.xxx. Subnet masks define how many bits of the IP address are used to identify the network.

By using 1s in all the address bits that identify the network, the subnet mask shows you which bits define the network and which are internal to the network. In the figure, the subnet mask 255.255.255.0 indicates that the first three octets in the address define the network.

Gateway IP

Enter one address for the gateway server or router that maintains a more complete listing of the surrounding networks. Messages that are destined for machines outside the local network are sent to the gateway address for address resolution.

Domain name server IP

Enter the address of the server for the domain in which the target is a member.

Domain name

Enter the name for the domain. Without the domain name, the target cannot communicate on the network within the domain.

Host name (less than 64 characters)

Enter the name of the host. Usually this value is the NetBIOS name for the machine if it exists.

See Also

C6000 TCP/IP Receive, C6000 TCP/IP Send,

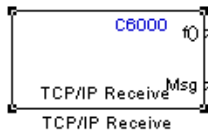
C6000 TCP/IP Receive

Receive message from remote IP interface

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ Target Communication

Description



Adding this block to your Simulink model results in generated code that configures TCP/IP on your target to receive messages.

To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

The block receives the message from the specified IP address on a host machine and passes it out the Msg port to a downstream block. The size of the message is unrestricted.

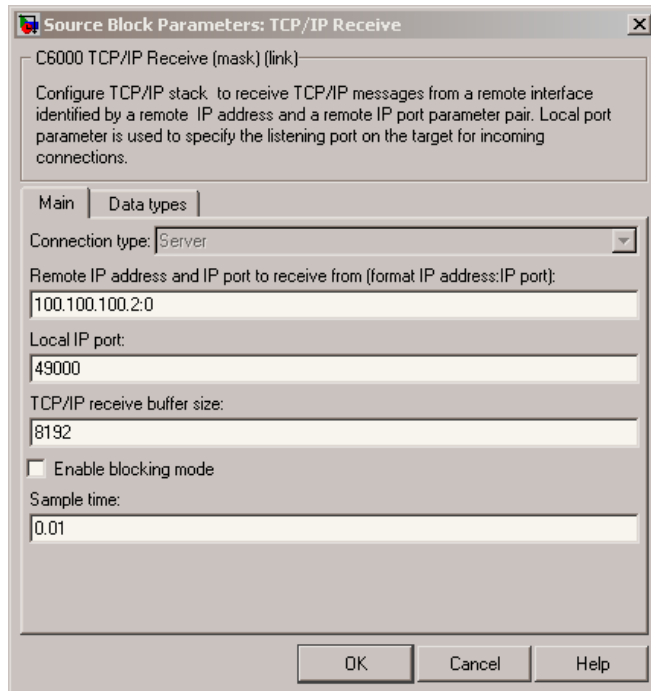
A second block output is a function call port that issues a function call whenever a new message is available on the receive buffer.

In simulations, this block outputs a stream of data (default type `uint8_T`) from the Msg port with the first bytes set to `0xFF` and the rest set to `0x00`. When the function call port exists, it generates a function call for every sample time hit.

Models that contain this block generate code for the parameters that configure TCP/IP on the target, including the ports, buffers, and message sizes.

Dialog Box

Main Pane



Connection type

Connection type specifies the connection initiation method used for the block. This is a read-only parameter — you cannot change it.

A **Server** connection creates a listening socket at the IP address and port in **Local IP port**. The TCP/IP layer uses this socket to accept incoming connection requests. External TCP/IP interfaces that send TCP/IP data to this block must actively seek the connection to establish communications (the *client* model).

Remote address and IP port to receive from (format IP Address:IP port)

Identifies the remote TCP/IP interface, by IP address and IP port, from which the block expects to receive messages. The input format uses the IP address and IP port identifier, separated by a colon. IP port value ranges from 0 to 65535. Entering a 0

for the IP port when the **Connection type** is **Client** specifies that the TCP/IP stack automatically assigns a port to use to seek connections.

Local IP port

This option identifies the IP port to use when **Connection type** is **Server** and when it is **Client**.

When you choose **Server**, **Local IP port** specifies the well-known port of the target TCP/IP server. Your IP port value must lie between 1 and 65535.

When you specify **Client** for the connection type, **Local IP port** specifies the TCP/IP address for the client socket. The IP port value can range from 0 to 65535, where 0 specifies that the TCP/IP stack assigns an ephemeral port automatically to seek connections.

TCP/IP receive buffer size

Specifies the size of the buffer used for queuing incoming TCP/IP messages. Typically, larger TCP/IP receive buffers provide a cushion for packet drops and can improve efficiency. The compiler allocates the TCP/IP receive buffer on the heap.

all TCP/IP blocks that specify a common local IP port must share a common TCP/IP receive buffer, because the size of the TCP/IP buffer is set only for the listening socket. all active connecting sockets inherit their buffer size value from the listening socket.

Enable blocking mode

Select this option to put the calling TCP/IP task into blocking mode so that the block receives messages completely before outputting the messages in the buffer to downstream blocks. Blocks connected to the receive block do not execute until the receive process completes. In blocking mode, program execution for receiving data stops until data in the message buffer is received.

Clearing this option puts the block in non blocking mode. The block checks the number of bytes in the TCP/IP receive buffer and returns output data only when the receive buffer contains more data than requested.

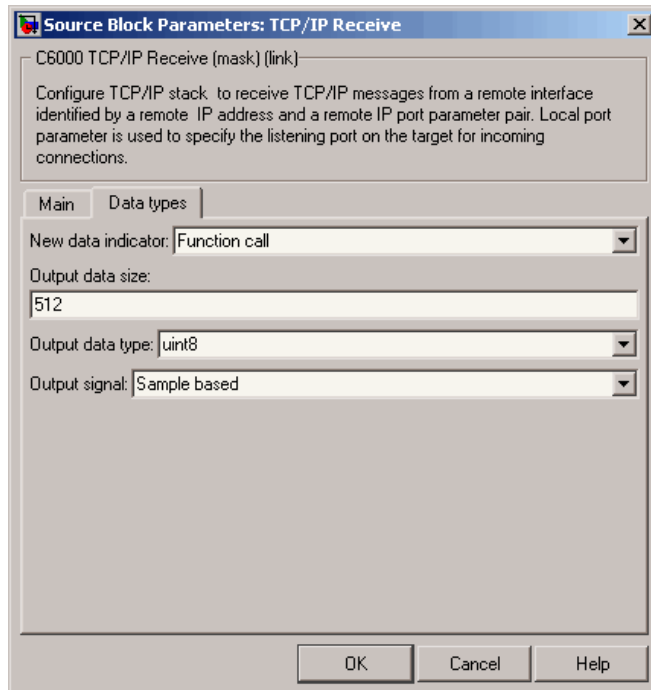
The block receives or outputs data continuously. Processes do not wait for data. Disabling blocking activates the **Sample time** parameter and adds an additional function call port to the block that indicates when the data port contains new, valid data.

Selecting blocking mode activates the **Timeout** parameter.

Sample Time

Use this option to specify when the block polls for new messages. This parameter value should be positive. Setting this to a specific value, often large, can reduce the chances of TCP/IP messages getting dropped. The default sample time is 0.01 seconds.

Data Types Pane



New Data Indicator

Use this option to specify how new data is indicated, either by a function call or a Boolean status.

Output Data Size

Use this option to specify the size of the output data, the units depend on the output data type.

Output Data Type

Use this option to specify the type of the output data. The value selected can be a built-in Simulink data type.

Output Signal

Use this option to specify whether the output signal is to be frame-based or sample-based.

See Also

C6000 TCP/IP Send, C6000 UDP Receive, C6000 UDP Send

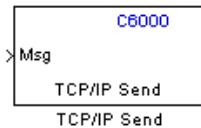
C6000 TCP/IP Send

Send message to remote IP interface

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ Target Communication

Description



Adding this block to your Simulink model results in generated code that configures TCP/IP on your target to send messages.

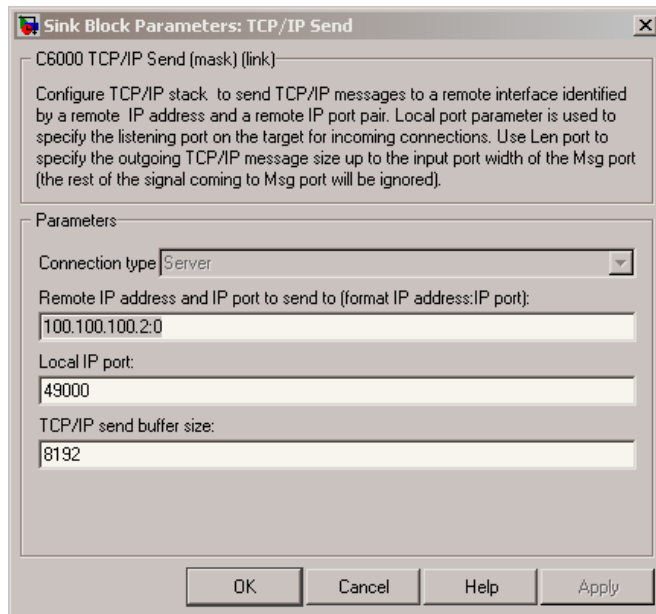
To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

The block sends the message to the specified IP address on a host machine. The data type of the message is unrestricted, as long as it is a built-in Simulink data type. The size of the data to be transmitted is also unrestricted.

Models that contain this block generate code for the parameters that configure TCP/IP on the target, including the ports, buffers, and message sizes.

Dialog Box



Connection type

Connection type specifies the connection initiation method used for the block. This is a read-only parameter — you cannot change it.

A **Server** connection creates a listening socket at the IP address and port in **Local IP port**. The TCP/IP layer uses this socket to accept incoming connection requests. For an external TCP/IP interface to receive TCP/IP data from this block, it must actively seek the connection to establish communications (the *client* model).

IP Address:IP port). External interfaces that want to exchange data with this block must be listening at the specified remote IP address and port.

Remote IP address and IP port to send to (format IP address:IP port)

Identifies the remote TCP/IP interface, by IP address and IP port, to which the block expects to send messages. The input format uses the IP address and IP port identifier, separated by a colon. IP port value ranges from 0 to 65535. Entering a 0 for the IP port when the **Connection type** is **Client** specifies that the TCP/IP stack automatically assigns a port to use to seek connections.

Local IP port

This option identifies the IP port used when **Connection type** is **Server**.

When the connection type is **Server**, **Local IP port** specifies the well-known port of the target TCP/IP server. The IP port value must lie between 1 and 65535.

TCP/IP send buffer size

Specifies the size of the buffer used for queuing outgoing TCP/IP messages. Typically, larger TCP/IP receive buffers provide a cushion for packet drops and can improve efficiency. The compiler allocates the TCP/IP send buffer on the heap.

all TCP/IP blocks that specify a common local IP port must share a common TCP/IP send buffer, because the size of the TCP/IP buffer is set only for the listening socket. all active connecting sockets inherit their buffer size value from the listening socket.

See Also

C6000 TCP/IP Receive, “UDP Send”, “UDP Receive”

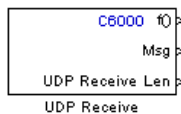
C6000 UDP Receive

Receive uint8 vector as UDP message

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ Target Communication

Description



This block configures the Ethernet driver on the target to receive UDP messages. A UDP message comes into this block from the transport layer, usually TCP/IP. The block passes the message to the next downstream block out the Msg port. One block output (Msg) is the data vector from the message. A second output is a flag that indicates when a new UDP message is available. A third output specifies the length of the message for variable length messages.

To use this block with the C6416, or C6713 DSK targets, you must meet the following requirements.

- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

This block reads a single UDP packet every sample hit. It does not attempt to receive multiple UDP packets to fill the output vector. If the UDP packet size is greater than the output port width parameter, UDP messages at the Msg port are truncated. The part for the UDP packet that does not fit into the Msg port is discarded as a result. The missing message content cannot be retrieved. Conversely, if the UDP packet size is smaller than the Msg port width specified, the portion of the output vector that does not fit into the specified size is invalid data.

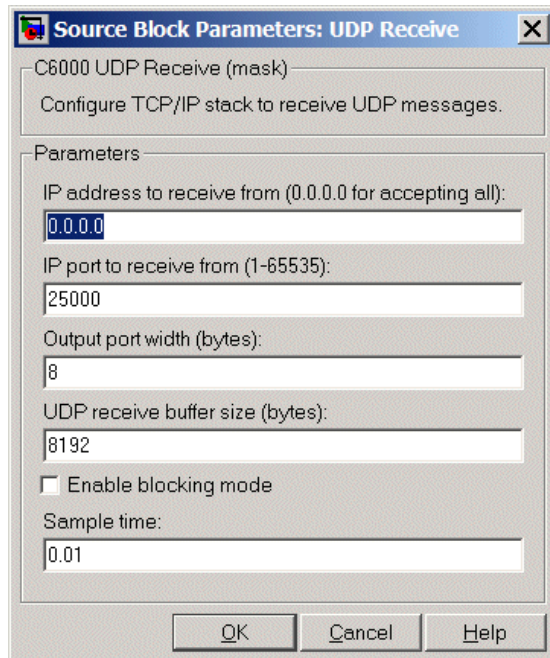
In non blocking mode, the data in the Msg port is not valid unless the block issues a function call.

C6000 UDP Receive blocks operate only to generate code for the target Ethernet driver. They do not perform a function in simulation and their simulation outputs are zeros.

Note To use the C6000 UDP Send and C6000 UDP Receive blocks, you must include the C6000 IP Config block to configure the Ethernet parameters for the target network. This block sets up the IP drivers for use and must be in the model for network-related processing.

Additional options let you decide whether the UDP messages work in blocking mode and set the sampling time for polling for new messages.

Dialog Box



IP address to receive from (0.0.0.0 to accept all)

Specifies the IP address from which the block accepts messages. Setting the address 0.0.0.0 configures the block to accept messages from all IP addresses. Setting a specific address, not 0.0.0.0, directs the block to accept messages from the specified address only.

Selecting Enable blocking mode, disables the **IP address to receive from** parameter. As a result, the block accepts messages from any IP address. You must clear **Enable blocking mode** to set this parameter to a specific IP address. The block must be in non blocking mode to specify the address to receive messages from via UDP.

IP port to receive from

Specify the port on this machine from which the block accepts messages. The other end of the communication, usually a “UDP Send” block, sends messages to this port. The value defaults to 25000, but the values can range from 1 to 65535.

Output port width (bytes)

Specifies the width of messages that the block accepts. When you design the transmit end of the UDP communication channel, you decide the message width. Set this parameter to a value equal or greater than the size of messages you expect to receive.

UDP receive buffer size (bytes)

Specify the size of the buffer in which UDP messages are stored when received. 8192 bytes is the default size. You need a buffer large enough to store UDP messages that come in while your process reads a message from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Enable blocking mode

Select this option to put the UDP receive process in blocking mode meaning the block outputs received messages before accepting input new messages. In blocking mode, program execution for receiving data stops until data in the buffer is sent. In non blocking mode, the block can receive or send data continuously. Processes do not wait for data.

Sample time (seconds)

Use this option to specify when the block polls for new messages. The value entered here should be greater than zero. Setting this to a specific value, often large, can reduce the chances of UDP messages getting dropped. The default sample time is 0.01 seconds.

See Also

C6000 TCP/IP Receive, C6000 TCP/IP Send, C6000 UDP Send

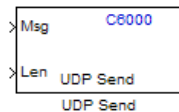
C6000 UDP Send

Send UDP message to host

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ Target Communication

Description



The UDP send block configures the target's on-board Ethernet driver to receive a `uint8` vector that it sends as a UDP message to the host. Models can contain only one C6000 UDP Send block.

To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

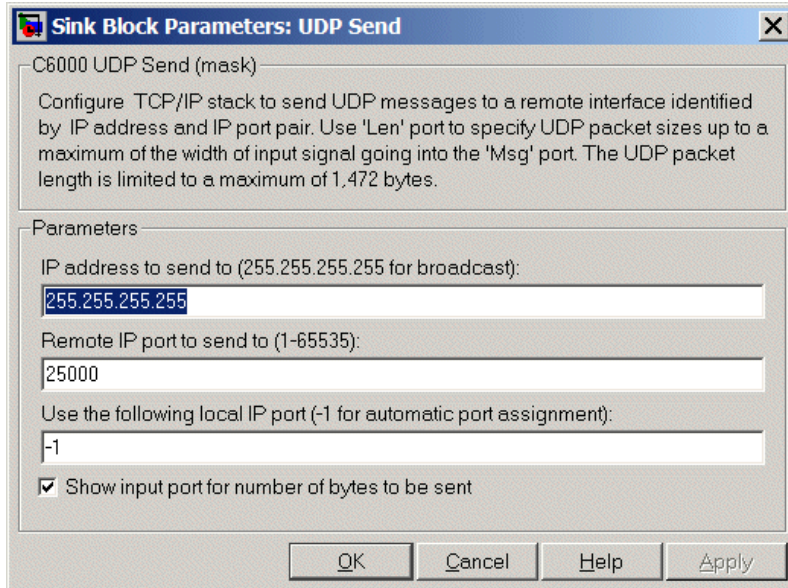
- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

Msg input format must be a `uint8` vector with UDP format. To use variable length messages, supply the message length for each message as input to the Len port. Message length can be an integer value in bytes up to the input width of signal at the Msg port.

C6000 UDP Send blocks operate only to generate code for the target Ethernet driver. They do not perform a function during simulation and they output zero.

Note To use the UDP Send and Receive blocks, for network processing, you must include the C6000 IP Config block to set up the IP drivers for the target Ethernet network.

Dialog Box



IP address to send to (255.255.255.255 for broadcast)

Specify the IP address to which the block sends the message. If you enter the address 255.255.255.255, the block broadcasts message to a listening IP address. If you enter a specific IP address, you limit the block to sending the message to the specified address.

Remote IP port to send to (1–65535)

Specify the port on the host to which the block sends the message. Port numbers range from 1 to 65535.

Note This port designation must match the port number where you configure the host to receive UDP messages.

Use the following local IP port (–1 for automatic port assignment)

Specify the local IP port the block sends the message from. If you accept the default value of –1, the network automatically selects the local IP port for sending the message.

If the address you are sending to expects the message to come from a specific port, enter that port address in this parameter. If you entered a port number in the “UDP Receive” block option **Remote IP port to receive from**, enter that port identifier in this parameter also.

Show input port for the number of bytes to be sent

Adds a block input port that lets you specify the number of bytes to send for each UDP message. The maximum allowed value is 1472 bytes. Use the input to dynamically the change the length of each message.

See Also

C6000 TCP/IP Receive, C6000 TCP/IP Send, C6000 UDP Receive

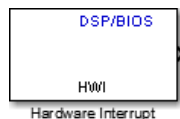
DSP/BIOS Hardware Interrupt

Generate Interrupt Service Routine

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

Description



Creates an Interrupt Service Routine (ISR) that executes the task block or subsystem that is downstream from the block. ISRs are functions that the CPU executes in response to an external event.

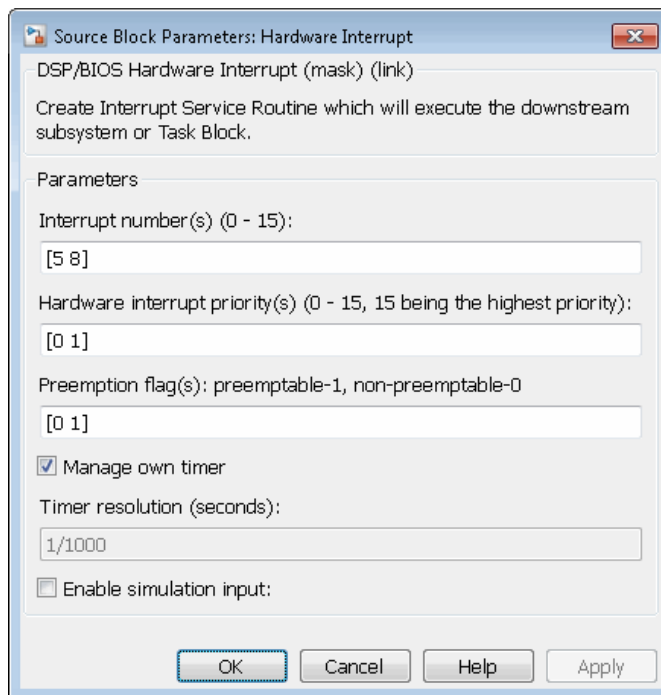
Interrupt numbers for C6000 family processors range from 0 to 15, with 0 reserved for the reset ISR. The following table presents the set of interrupt numbers for the C6713 processor. For more detailed and specific information about interrupts, refer to Texas Instruments technical documentation for your target processor.

Interrupt Number	Default Event	Module
0	Reset	
1	NMI	
2	Reserved	
3	Reserved	
4	GPINT4	GPIO
5	GPINT5	GPIO
6	GPINT6	GPIO
7	GPINT7	GPIO
8	EDMAINT	EDMA

Interrupt Number	Default Event	Module
9	EMUDTDMA	Emulation
10	SDINT	EMIF
11	EMURTDXR	Emulation
12	EMURTDXTX	Emulation
13	DSPINT	HPI
14	TINT0	Timer 0
15	TINT1	Timer 1

In models, you usually follow this block with either a DSP/BIOS Task or DSP/BIOS Triggered Task block.

Dialog Box



Interrupt number(s)

Enter one or more integer values as a vector that represent interrupts. Interrupts have a value from 0, the highest priority to 15, lowest priority. As shown, enter the values enclosed in square brackets. For example, entering

```
[3 5 15]
```

results in three interrupt routines. [5 8] is the default entry, specifying two interrupts.

Preemption flag(s)

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value here, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 5 in **Interrupt numbers** is not preemptible and the priority 8 interrupt can be preempted.

Manage own timer

The ISR generated by the this block can manage its own time by reading time from the clock on the board. Selecting this option directs the ISR to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the ISR uses.

Timer resolution (seconds)

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

Enable simulation input

Selecting this option adds an input port to the block for simulating inputs in Simulink software. Connect interrupt simulation sources to the input. This option affects simulation only. It does not alter generated code.

See Also

DSP/BIOS Task, DSP/BIOS Triggered Task

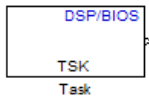
DSP/BIOS Task

Create task that runs as separate DSP/BIOS thread

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

Description

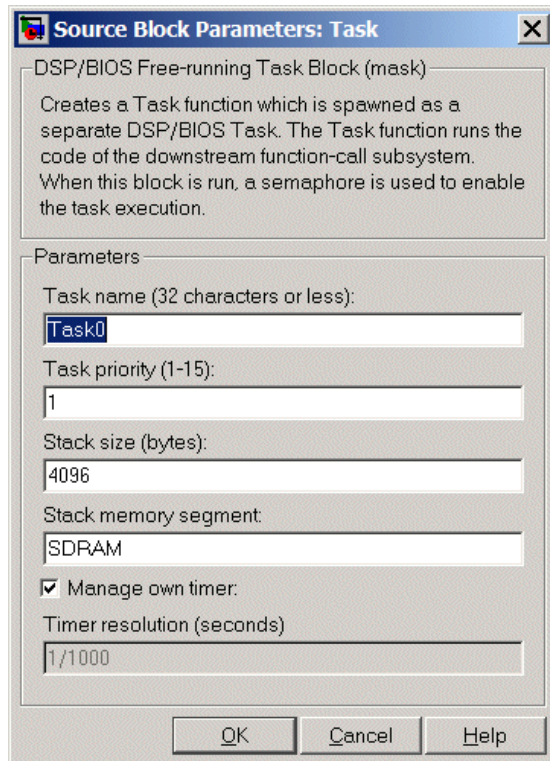


Creates a free-running task that runs in response to an ISR and as a separate DSP/BIOS™ thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_ert.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

Dialog Box



Task name (32 characters or less)

Creates a name for the task. Enter a string of up to 32 characters, including numbers and letters. You cannot use the standard C reserved characters, such as / and : in the name.

Task priority (1-15)

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority.

Stack size (bytes)

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Each DSP/BIOS task has a separate stack. This parameter is not related to **System stack size (MAUs)** in the model Configuration Parameters.

Stack memory segment

Specify where the stack resides in memory.

Manage own timer

This block can manage its own time by reading time from the clock on the board. Selecting this option directs the task/block to maintain the time itself. When you select **Manage own timer**, you enable the **Timer resolution** option that reports the timer resolution the task uses.

Timer resolution (seconds)

When you direct the block to manage its own time, this option (available only when you select **Manage own timer**) reports the resolution of the clock. **Timer resolution** is a read-only parameter. You cannot change the value.

See Also

DSP/BIOS Hardware Interrupt, DSP/BIOS Triggered Task

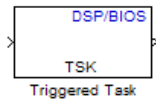
DSP/BIOS Triggered Task

Create asynchronously triggered task

Library

Embedded Coder Support Package for Texas Instruments C6000 Processors/ DSP/BIOS

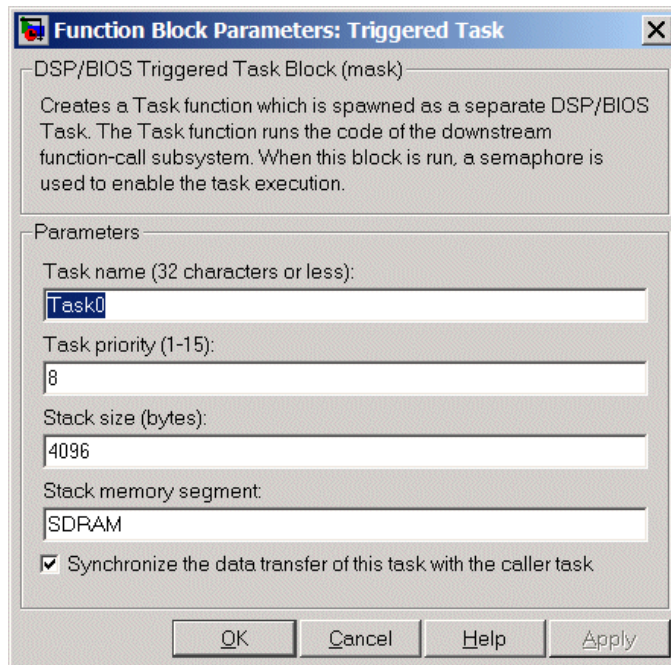
Description



Creates a task that runs asynchronously in response to an ISR and as a separate DSP/BIOS thread. The spawned task runs the downstream function call subsystem in the model.

When the process runs this task, it uses a semaphore structure to enable the task and restrict access by it to other resources.

Dialog Box



Task name (32 characters or less)

Creates a name for the task. Enter a string of up to 32 characters, including numbers and letters. You cannot use the standard C reserved characters, such as / or : in the name.

Task priority (1-15)

Sets the priority for the task, where 1 is the lowest priority and 15 the highest. Higher priority tasks can preempt tasks that have lower priority, unless the preemptible flag (**Preemption flag** option on the “C5000/C6000 Hardware Interrupt” block) prevents preempting the task.

Stack size (bytes)

Specify the size of the stack the task uses. The value defaults to 4096 bytes. Take care to set this value to a value that is large enough. If the task uses more than the allotted space it can write into other memory areas with unintended results.

Each DSP/BIOS task has a separate stack. This parameter is not related to **System stack size (MAUs)** in the model Configuration Parameters.

Stack memory segment

Specify where the stack resides in memory by specifying the memory segment. Additional information about DSP/BIOS memory segments also appears in the Target Hardware Resources tab.

Synchronize data transfer of this task with caller task

Specify whether this task should synchronize data transfer with the calling task. Select this option to enable synchronization. Clearing this option enables the **Timer resolution** option.

Timer resolution

When you direct the block not to synchronize data with the calling task (by clearing **Synchronize data transfer of this task with caller task**), **Timer resolution** reports the resolution of the timer. **Timer resolution** is a read-only parameter. You cannot change the value.

See Also

DSP/BIOS Hardware Interrupt, DSP/BIOS Task

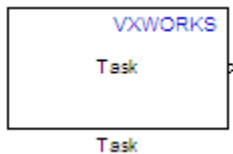
VxWorks Task

Spawn task function as separate VxWorks thread

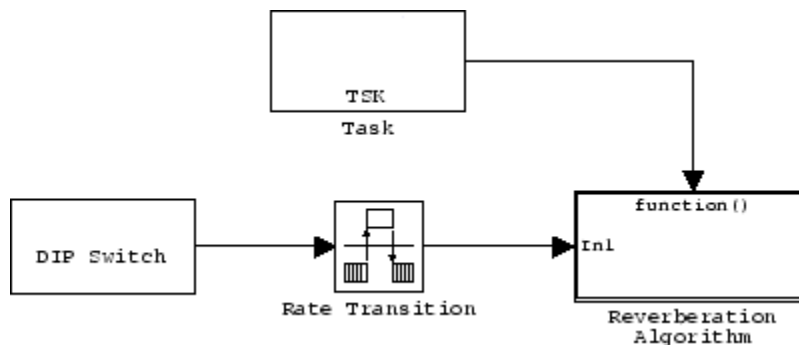
Library

Embedded Coder Support Package for Wind River® VxWorks® RTOS

Description



Use this block to create a task function that spawns as a separate VxWorks thread. The task function runs the code of the downstream function-call subsystem. For example:

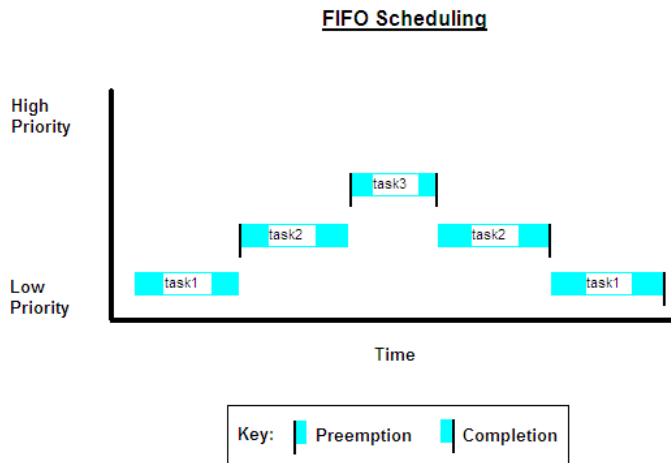


In order to use this block, set the **System target file** parameter to `idelink_ert.tlc` or `idelink_grt.tlc`. The **System target file** parameter is located on the Code Generation pane of the Model Configuration Parameters dialog, which you can view by selecting your model and pressing **Ctrl+E**.

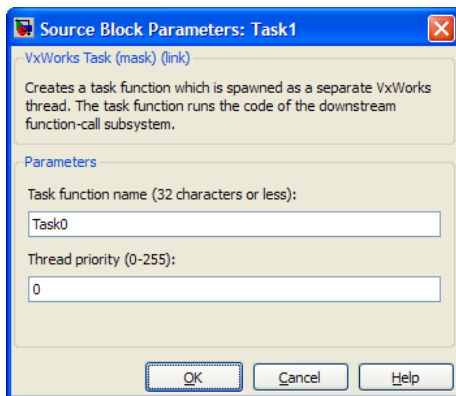
The VxWorks Task block uses a First In, First Out (FIFO) scheduling algorithm, which executes real-time processes without time slicing. With FIFO scheduling, a higher-

priority process preempts a lower-priority process. While the higher-priority process runs, the lower-priority process remains at the top of the list for its priority. When the scheduler blocks the higher-priority processes, the lower-priority process resumes.

For example, in the following image, task2 preempts task1. Then, task3 preempts task2. When task3 completes, task2 resumes. When task2 completes, task1 resumes.



Dialog



Task name

Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

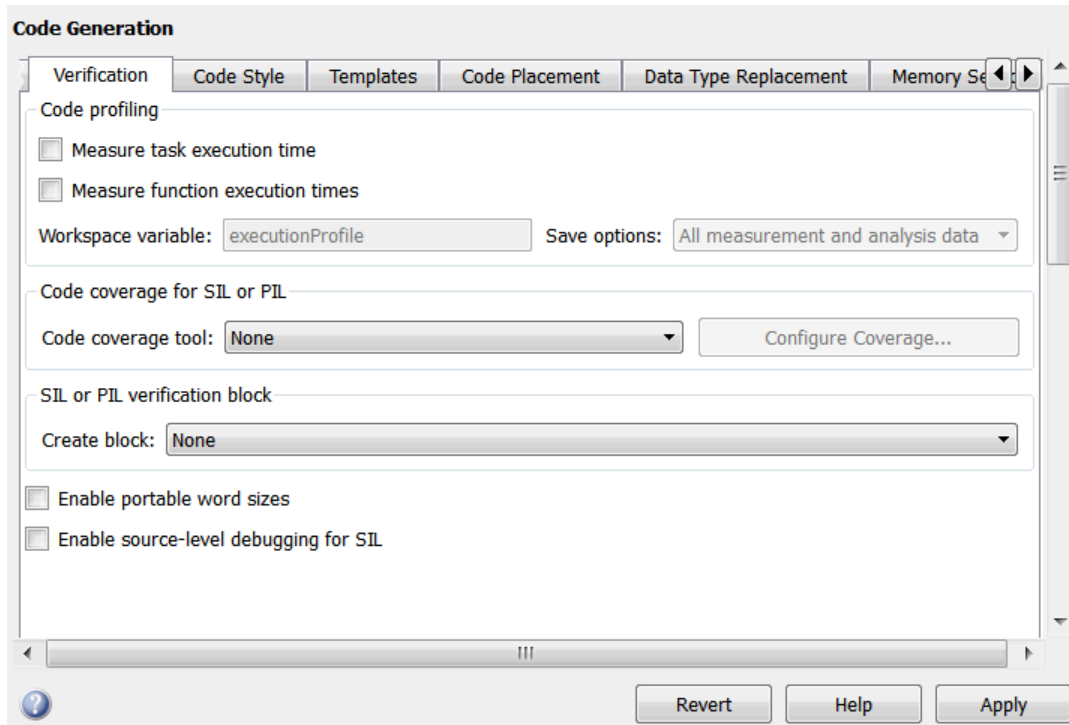
Thread priority (0 to 255)

Set the priority for the thread, from 0 to 255 (low-to-high). Higher-priority tasks can preempt lower-priority tasks.

Configuration Parameters

- “Code Generation Pane: Verification” on page 3-2
- “Code Generation Pane: Code Style” on page 3-13
- “Code Generation Pane: Templates” on page 3-26
- “Code Generation Pane: Code Placement” on page 3-35
- “Code Generation Pane: Data Type Replacement” on page 3-47
- “Code Generation Pane: Memory Sections” on page 3-68
- “Code Generation Pane: AUTOSAR Code Generation Options” on page 3-79
- “Code Generation: Coder Target Pane” on page 3-83
- “Code Generation: Target Hardware Resources Pane” on page 3-107
- “Coder Target Pane: Altera Cyclone V SoC development kit, Arrow SoCKit development board” on page 3-177
- “Coder Target Pane: ARM Cortex-A9 (QEMU)” on page 3-181
- “Coder Target Pane: ARM Cortex-M3 (QEMU)” on page 3-185
- “Coder Target Pane: Embedded Coder Support Package for Freescale FRDM-KL25Z Board” on page 3-188
- “Coder Target Pane: BeagleBone Black Hardware” on page 3-193
- “Coder Target Pane: Support Package for STMicroelectronics STM32F4 Discovery Hardware” on page 3-198
- “Coder Target Pane: Texas Instruments C2000 Processors” on page 3-210
- “Coder Target Pane: Texas Instruments Concerto F28M3x (ARM Cortex-M3)” on page 3-270
- “Coder Target Pane: Xilinx Zynq ZC702/ZC706 Evaluation Kits, ZedBoard” on page 3-277
- “Parameter Reference” on page 3-281

Code Generation Pane: Verification



In this section...

“Code Generation: Verification Tab Overview” on page 3-3

“Measure task execution time” on page 3-3

“Measure function execution times” on page 3-4

“Workspace variable” on page 3-5

“Save options” on page 3-6

“Code coverage tool” on page 3-7

“Create block” on page 3-9

“Enable portable word sizes” on page 3-10

“Enable source-level debugging for SIL” on page 3-11

Code Generation: Verification Tab Overview

Create SIL block and configure word size portability, code coverage for SIL testing, and code execution profiling

Configuration

This tab appears only if you specify an ERT-based system target file.

See Also

“About SIL and PIL Simulations”

Measure task execution time

Specify whether to collect execution time profiles for tasks in generated code

Settings

Default: off

On

Collect measurements of execution times. For SIL and PIL simulations, the software obtains data from instrumentation probes in the SIL or PIL test harness code.

Off

Do not collect measurements of execution times

Dependencies

When you use this parameter, you must also specify a workspace variable. The software uses this variable to collect execution time measurements.

In a model reference hierarchy, the top-model parameter value applies to the whole hierarchy. The software ignores the value of this parameter in referenced models.

Command-Line Information

Parameter: CodeExecutionProfiling

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	Off

See Also

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Measure function execution times

Specify whether to collect execution times for functions inside generated code

Settings

Default: off

On

Collect execution times for functions. Data obtained from instrumentation probes placed inside code generated from atomic subsystems and model reference hierarchies.

Off

Do not collect execution times for functions inside generated code

Dependencies

To use this parameter, you must also select **Measure task execution time**. If the current model is a referenced model, you must select **Measure task execution time** for the top model of the model reference hierarchy.

For a model in a reference hierarchy, the software does not support simultaneous function execution time measurement and code coverage.

Command-Line Information**Parameter:** CodeProfilingInstrumentation**Type:** string**Value:** 'on' | 'off'**Default:** 'off'**Recommended Settings**

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	Off

See Also

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Workspace variable

Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles

Settings**Default:** executionProfile

When you run simulation, software generates specified workspace variable as an `coder.profile.ExecutionTime` object. To view and analyze execution profiles, use methods from the `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` classes.

Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Command-Line Information

Parameter: CodeExecutionProfileVariable

Type: string

Value: valid MATLAB variable name

Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Valid MATLAB variable name
Efficiency	No impact
Safety precaution	No impact

See Also

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Save options

Specify whether to save code profiling measurement and analysis data to base workspace

Settings

Default: Summary data only

Summary data only

Save only code profiling summary data to a `coder.profile.ExecutionTime` in the base workspace. Use this option to limit the amount of data that the software saves to base workspace. For example, if you are concerned that your computer may not have enough memory to store the time measurements for a long simulation. The software calculates metrics for the code execution report as the simulation proceeds, without saving raw data to memory. To view these metrics, use the `coder.profile.ExecutionTime` report method.

All measurement and analysis data

Save the code profiling measurement and analysis data to a `coder.profile.ExecutionTime` object in the base workspace. In addition to viewing the code execution report, this option allows you to analyze data using `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` methods.

Dependency

You can only specify this parameter if you select the **Measure task execution time** check box. Otherwise the field appears dimmed.

Command-Line Information

Parameter: `CodeProfilingSaveOptions`

Type: string

Value: 'SummaryOnly' | 'AllData'

Default: 'SummaryOnly'

Recommended Settings

Application	Setting
Debugging	All measurement and analysis data
Traceability	All measurement and analysis data
Efficiency	Summary data only
Safety precaution	No impact

See Also

- “Configure Code Execution Profiling for SIL and PIL”
- “View and Compare Code Execution Times”
- “Analyze Code Execution Data”

Code coverage tool

Specify a code coverage tool

Settings

Default: None

None

No code coverage tool specified

BullseyeCoverage

Specifies the BullseyeCoverage™ tool from Bullseye Testing Technology™

LDRA Testbed

Specifies the LDRA Testbed® tool from LDRA Software Technology

Dependencies

You cannot specify this parameter if **Create block** is either **SIL** or **PIL**.

If you do not specify a tool, **Configure Coverage** appears dimmed. If you specify a tool, click **Configure Coverage** to open the Code Coverage Settings dialog box.

Command-Line Information

Parameter: CoverageTool

Type: string

Value: 'None' | 'BullseyeCoverage' | 'LDRA Testbed'

Default: 'None'

Tip To access the CoverageTool parameter, type:

```
covSettings = get_param(gcs, 'CodeCoverageSettings');  
covSettings.CoverageTool
```

Recommended Settings

Application	Setting
Debugging	BullseyeCoverage or LDRA Testbed
Traceability	BullseyeCoverage or LDRA Testbed
Efficiency	None (code coverage off)
Safety precaution	None (code coverage off)

See Also

- “Configure SIL and PIL Code Coverage”
- “Configure Code Coverage Programmatically”

Create block

Generate a SIL or PIL block

Settings

Default: None

None

SIL or PIL block not generated.

SIL

Generate a SIL block that represents a top-model or subsystem.

If you select this option, the software creates and opens an untitled model with a SIL block. The SIL block contains an S-function wrapper, through which the software runs compiled object code on the host computer. With this block, you can verify the behavior of source code generated from top-model or subsystem components.

PIL

Generate a PIL block that represents a top-model or subsystem.

If you select this option, the software creates and opens an untitled model with a PIL block. The PIL block contains an S-function, through which the software runs cross-compiled object code on a target processor or instruction set simulator. With this block, you can verify the behavior of object code generated from top-model or subsystem components.

To control the way code compiles and executes in the target environment, use Target Connectivity API.

Command-Line Information

Parameter: CreateSILPILBlock

Type: string

Value: 'None' | 'SIL' | 'PIL'

Default: 'None'

Recommended Settings

Application	Setting
Debugging	On

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

See Also

- “Use a SIL or PIL Block”
- Techniques for Exporting Function-Call Subsystems
- “About SIL and PIL Simulations”

Enable portable word sizes

Allow portability across host and target processors that support different word sizes.

You can enable portable word sizes to support SIL testing of your generated code. For a SIL simulation, select **SIL** in the **Create block** field, or use top-model or Model block SIL simulation mode.

Settings

Default: off

On

Generate conditional processing macros to support compilation of generated code on a processor that supports a different word size than the target processor on which production code is intended to run. This option allows you to use the same generated code for software-in-the-loop (SIL) testing on the host platform and production deployment on the target platform. For example, you can perform SIL testing on a 32-bit host and deploy the code on a 16-bit target.

Off

Does not generate portable code.

Dependencies

When you use this option, you should select **Test hardware is the same as production hardware** on the **Hardware Implementation** pane.

Command-Line Information**Parameter:** PortableWordSizes**Type:** string**Value:** 'on' | 'off'**Default:** 'off'**Recommended Settings**

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

See Also

“Configure Hardware Implementation Settings for SIL”

Enable source-level debugging for SIL

Allow debugging of generated code during a SIL simulation

Settings**Default:** off
 On

Source-level debugging is enabled.

 Off

Source-level debugging is disabled.

Command-Line Information**Parameter:** SILDebugging**Type:** string**Value:** 'on' | 'off'**Default:** 'off'

Recommended Settings

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

See Also

“Debug Code During SIL Simulations”

Code Generation Pane: Code Style

The screenshot shows the 'Code style' configuration window. It is divided into two main sections: 'Code style' and 'Code indentation'.
In the 'Code style' section, there is a 'Parentheses level:' dropdown menu set to 'Nominal (Optimize for readability)'. Below it are five checkboxes: 'Preserve operand order in expression' (unchecked), 'Preserve condition expression in if statement' (unchecked), 'Convert if-elseif-else patterns to switch-case statements' (unchecked), 'Preserve extern keyword in function declarations' (checked), and 'Suppress generation of default cases for Stateflow switch statements if unreachable' (unchecked). At the bottom of this section is a 'Casting Modes:' dropdown menu set to 'Nominal'.
The 'Code indentation' section contains an 'Indent style:' dropdown menu set to 'K&R' and an 'Indent size:' dropdown menu set to '2'.

In this section...

“Code Generation: Code Style Tab Overview” on page 3-13

“Parentheses level” on page 3-14

“Preserve operand order in expression” on page 3-15

“Preserve condition expression in if statement” on page 3-16

“Convert if-elseif-else patterns to switch-case statements” on page 3-17

“Preserve extern keyword in function declarations” on page 3-19

“Suppress generation of default cases for Stateflow switch statements if unreachable” on page 3-20

“Casting Modes” on page 3-21

“Indent style” on page 3-23

“Indent size” on page 3-24

Code Generation: 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

- “Control Code Style”
- “Code Generation Pane: Code Style” on page 3-13

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 ANSI¹ C or C++, or to override default precedence. For example:

```
Out = In2 - In1 > 1.0 && In2 > 2.0;
```

Nominal (Optimize for readability)

Inserts parentheses in a way that compromises between readability and visual complexity. For example:

```
Out = ((In2 - In1 > 1.0) && (In2 > 2.0));
```

Maximum (Specify precedence with parentheses)

Includes parentheses to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA^{®2} requirements. For example:

```
Out = (((In2 - In1) > 1.0) && (In2 > 2.0));
```

Command-Line Information

Parameter: ParenthesesLevel

Type: string

Value: 'Minimum' | 'Nominal' | 'Maximum'

Default: 'Nominal'

1. ANSI is a registered trademark of the American National Standards Institute, Inc.
2. MISRA is a registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.

Recommended Settings

Application	Setting
Debugging	Nominal (Optimized for readability)
Traceability	Nominal (Optimized for readability)
Efficiency	Minimum (Rely on C/C++ operators for precedence)
Safety precaution	Maximum (Specify precedence with parentheses)

See Also

Controlling Parenthesization

Preserve operand order in expression

Specify whether to preserve order of operands in expressions.

Settings

Default: off

On

Preserves the expression order specified in the model. Select this option to increase readability of the code or for code traceability purposes.

$A*(B+C)$

Off

Optimizes efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive. For example:

$(B+C)*A$

Command-Line Information

Parameter: PreserveExpressionOrder

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	On

Preserve condition expression in if statement

Specify whether to preserve empty primary condition expressions in `if` statements.

Settings

Default: off

On

Preserves empty primary condition expressions in `if` statements, such as the following, to increase the readability of the code or for code traceability purposes.

```
if expression1
else
    statements2;
end
```

Off

Optimizes empty primary condition expressions in `if` statements by negating them. For example, consider the following `if` statement:

```
if expression1
else
    statements2;
end
```

By default, the code generator negates this statement as follows:

```
if ~expression1
    statements2;
end
```


Command-Line Information**Parameter:** PreserveIfCondition**Type:** string**Value:** 'on' | 'off'**Default:** 'off'**Recommended Settings**

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	On

Convert if-elseif-else patterns to switch-case statements

Specify whether to generate code for if-elseif-else decision logic as switch-case statements.

This readability optimization works on a per-model basis and applies only to:

- Flow charts in Stateflow[®] charts
- MATLAB functions in Stateflow charts
- MATLAB Function blocks in that model

Settings**Default:** off
 On

Generate code for if-elseif-else decision logic as switch-case statements.

For example, assume that you have the following logic pattern:

```
if (x == 1) {
    y = 1;
} else if (x == 2) {
    y = 2;
} else if (x == 3) {
```

```
        y = 3;
    } else {
        y = 4;
    }
```

Selecting this check box converts the `if-elseif-else` pattern to the following `switch-case` statements:

```
switch (x) {
    case 1:
        y = 1; break;
    case 2:
        y = 2; break;
    case 3:
        y = 3; break;
    default:
        y = 4; break;
}
```

Off

Preserve `if-elseif-else` decision logic in generated code.

Command-Line Information

Parameter: ConvertIfToSwitch

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Off
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	No impact

See Also

- “Enhance Readability of Code for Flow Charts”
- “Enhance Code Readability for MATLAB Function Blocks”
- “Control Code Style”

Preserve extern keyword in function declarations

Specify whether to include the `extern` keyword in function declarations in the generated code.

Note: The `extern` keyword is optional for functions with external linkage. It is considered good programming practice to include the `extern` keyword in function declarations for code readability.

Settings

Default: on

On

Include the `extern` keyword in function declarations in the generated code. For example, the generated code for the model `rtwdemo_hyperlinks` contains the following function declarations in `rtwdemo_hyperlinks.h`:

```
/* Model entry point functions */
extern void rtwdemo_hyperlinks_initialize(void);
extern void rtwdemo_hyperlinks_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage. The function definitions in this example are in the generated file `rtwdemo_hyperlinks.c`.

Off

Remove the `extern` keyword from function declarations in the generated code.

Command-Line Information

Parameter: `PreserveExternInFcnDecls`

Type: string

Value: `'on' | 'off'`

Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact

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

See Also

For more information on code style options, see “Code Generation Pane: Code Style” on page 3-13.

Suppress generation of default cases for Stateflow switch statements if unreachable

Specify whether to generate default cases for switch-case statements in the code for Stateflow charts. This optimization works on a per-model basis. It applies to the code generated for a state that has multiple substates. For a list of the state functions in the generated code, see “Inline State Functions in Generated Code” in the Stateflow documentation.

Settings

Default: off

On

Do not generate the default case when it is unreachable. This setting enables better code coverage because every branch in the generated code is falsifiable.

Off

Generate a default case whether or not it is reachable. This setting supports MISRA C[®] compliance and provides a backup in case of RAM corruption.

For example, when the state has a nontrivial `entry` function, the following default case appears in the generated code for the `during` function:

```
default:  
  entry_internal();  
  break;
```

In this case, the code marks the corresponding substate as active.

Command-Line Information**Parameter:** SuppressUnreachableDefaultCases**Type:** string**Value:** 'on' | 'off'**Default:** 'off'**Recommended Settings**

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	On (execution, ROM), No impact (RAM)
Safety precaution	Off

See Also

For more information on code style options, see “Code Generation Pane: Code Style” on page 3-13.

Casting Modes

Specify how the code generator casts data types for variables.

Settings**Default:** Nominal**Nominal**

Generate code that uses default C compiler data type casting.

```
void rtwdemo_rtwecintro_step(void)
{
    boolean_T rtb_equal_to_count;
    rtDWork.X++;
    rtb_equal_to_count = (rtDWork.X != 16);
    if (rtb_equal_to_count && (rtPrevZCSigState.Amplifier_Trig_ZCE != POS_ZCSIG))
    {
        rtY.Output = rtU.Input << 1;
    }
}
```

Standards Compliant

Generate code that casts data types to conform to MISRA standards.

```
void rtdemo_rtweintro_step(void)
{
    boolean_T rtb_equal_to_count;
    rtDWork.X++;
    rtb_equal_to_count = (boolean_T)(int32_T)((int32_T)rtDWork.X != (int32_T)16);
    if (((int32_T)rtb_equal_to_count) && (rtPrevZCSigState.Amplifier_Trig_ZCE !=
        POS_ZCSIG)) {
        rtY.Output = (int32_T)(uint32_T)((uint32_T)rtU.Input << (uint32_T)(int8_T)1);
    }
}
```

Explicit

Generate code that casts data type values explicitly.

```
/* Model step function */
void rtdemo_rtweintro_step(void)
{
    boolean_T rtb_equal_to_count;
    rtDWork.X = (uint8_T)(1U + (uint32_T)(int32_T)rtDWork.X);
    rtb_equal_to_count = (boolean_T)((int32_T)rtDWork.X != 16);
    if (((int32_T)rtb_equal_to_count) && ((int32_T)((int32_T)
        rtPrevZCSigState.Amplifier_Trig_ZCE != (int32_T)POS_ZCSIG))) {
        rtY.Output = rtU.Input << 1;
    }
}
```

Command-Line Information

Parameter: CastingMode

Type: string

Value: 'Nominal' | 'Standards' | 'Explicit'

Default: 'Nominal'

Recommended Settings

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

See Also

For more information, see “Control Cast Expressions in Generated Code” and “MISRA C Guidelines”.

Indent style

Specify style for the placement of braces in generated code.

Settings

Default: K&R

K&R

For blocks within a function, an opening brace is on the same line as its control statement. For example:

```
void rt_OneStep(void)
{
    static boolean_T OverrunFlag = 0;
    if (OverrunFlag) {
        rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
        return;
    }

    OverrunFlag = TRUE;
    rtwdemo_counter_step();
    OverrunFlag = FALSE;
}
```

Allman

For blocks within a function, an opening brace is on its own line at the same level of indentation as its control statement. For example:

```
void rt_OneStep(void)
{
    static boolean_T OverrunFlag = 0;
    if (OverrunFlag)
    {
        rtmSetErrorStatus(rtwdemo_counter_M, "Overrun");
        return;
    }
}
```

```
    OverrunFlag = TRUE;  
    rtwdemo_counter_step();  
    OverrunFlag = FALSE;  
}
```

Command-Line Information

Parameter: IndentStyle

Type: string

Value: 'K&R' | 'Allman'

Default: 'K&R'

Recommended Settings

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

See Also

For more information on code indentation options, see “Control Indentation Style in Generated Code”.

Indent size

Specify indent size for generated code.

Settings

Default: 2

Specify an integer value that indicates the number of characters per indent level. Possible values range from 2–8 characters.

Command-Line Information

Parameter: IndentSize

Type: integer

Value: integer from 2–8

Default: 2

Recommended Settings

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

See Also

For more information on code indentation options, see “Control Indentation Style in Generated Code”.

Code Generation Pane: Templates

Code Generation

SIL and PIL Verification | Code Style | **Templates** | Code Placement | Data Type Replacement | Memory Sections

Code templates

Source file (*.c) template: ert_code_template.cgt [Browse...] [Edit...]

Header file (*.h) template: ert_code_template.cgt [Browse...] [Edit...]

Data templates

Source file (*.c) template: ert_code_template.cgt [Browse...] [Edit...]

Header file (*.h) template: ert_code_template.cgt [Browse...] [Edit...]

Custom templates

File customization template: example_file_process.tlc [Browse...] [Edit...]

Generate an example main program

Target operating system: BareBoardExample

In this section...

- “Code Generation: Templates Tab Overview” on page 3-26
- “Code templates: Source file (*.c) template” on page 3-27
- “Code templates: Header file (*.h) template” on page 3-27
- “Data templates: Source file (*.c) template” on page 3-28
- “Data templates: Header file (*.h) template” on page 3-29
- “File customization template” on page 3-30
- “Generate an example main program” on page 3-31
- “Target operating system” on page 3-33

Code Generation: 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

“Code Generation Pane: Templates” on page 3-26

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: valid CGT file

Default: `'ert_code_template.cgt'`

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Code templates: Header file (*.h) template

Specify the code generation template (CGT) file to use when generating a code header file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated header files (.h).

Note: The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTHdrFileBannerTemplate`

Type: string

Value: valid CGT file

Default: `'ert_code_template.cgt'`

Recommended Settings

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

See Also

- [Selecting and Defining Templates](#)
- [Custom File Processing](#)

Data templates: Source file (*.c) template

Specify the code generation template (CGT) file to use when generating a data source file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data source files (.c or .cpp) that contain definitions of variables of global scope.

Note: The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: ERTDataSrcFileTemplate

Type: string

Value: valid CGT file

Default: 'ert_code_template.cgt'

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Data templates: Header file (*.h) template

Specify the code generation template (CGT) file to use when generating a data header file.

Settings

Default: ert_code_template.cgt

You can use a CGT file to define the top-level organization and formatting of generated data header files (.h) that contain declarations of variables of global scope.

Note: The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: ERTDataHdrFileTemplate

Type: string

Value: valid CGT file

Default: 'ert_code_template.cgt'

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

File customization template

Specify the custom file processing (CFP) template file to use when generating code.

Settings

Default: 'example_file_process.tlc'

You can use a CFP template file to customize generated code. A CFP template file is a TLC file that organizes types of code (for example, includes, `typedefs`, and functions) into sections. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call a code template API to emit the buffered code into specified sections of generated source and header files. The CFP template file must be located on the MATLAB path.

Command-Line Information

Parameter: ERTCustomFileTemplate

Type: string

Value: valid TLC file

Default: 'example_file_process.tlc'

Recommended Settings

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

See Also

- Selecting and Defining Templates
- Custom File Processing

Generate an example main program

Control whether to generate an example main program for a model.

Settings

Default: on

On

Generates an example main program, `ert_main.c` (or `.cpp`). The file includes:

- The `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

The operation of the main program and the scheduling algorithm employed depend primarily on whether your model is single-rate or multirate, and also on your model's solver mode (`SingleTasking` or `MultiTasking`).

Off

Does not generate an example main program.

Note: The software provides static versions of the main file, `matlabroot/rtw/c/src/common/rt_main.c` and `matlabroot/rtw/c/src/common/`

`rt_cppclass_main.cpp`, as a basis for custom modifications. You can use either static main 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 main module.

Dependencies

- This parameter enables **Target operating system**.
- You must enable this parameter and select `VxWorksExample` for **Target operating system** if you use `VxWorks`³ 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

- “Generate a Standalone Program”

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

- Static Main Program Module
- Custom File Processing

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.

NativeThreadsExample

Generates a fully commented example showing how to deploy the threaded code under the host operating system. This option requires you to configure your model for concurrent execution.

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' | 'NativeThreadsExample'

Default: 'BareBoardExample'

Recommended Settings

Application	Setting
Debugging	No impact

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

See Also

- “Generate a Standalone Program”
- Static Main Program Module
- Custom File Processing

Code Generation Pane: Code Placement

Code Generation

SIL and PIL Verification | Code Style | Templates | **Code Placement** | Data Type Replacement | Memory Sections

Global data placement (custom storage classes only)

Data definition: Auto

Data declaration: Auto

#include file delimiter: Auto

Use owner from data object for data definition placement

Global data placement (MPT data objects only)

Signal display level: 10 | Parameter tune level: 10

Code Packaging

File packaging format: Modular

In this section...

- “Code Generation: Code Placement Tab Overview” on page 3-35
- “Data definition” on page 3-36
- “Data definition filename” on page 3-37
- “Data declaration” on page 3-38
- “Data declaration filename” on page 3-39
- “Use owner from data object for data definition placement” on page 3-40
- “#include file delimiter” on page 3-41
- “Signal display level” on page 3-42
- “Parameter tune level” on page 3-43
- “File packaging format” on page 3-44

Code Generation: Code Placement Tab Overview

Specify the data placement in the generated code.

Configuration

This tab appears only if you specify an ERT based system target file.

See Also

“Code Generation Pane: Code Placement” on page 3-35

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	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

“Data Definition and Declaration Management”

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 **Code Generation** pane: **Templates** tab.

If you specify C++ as the target language, omit the `.cpp` extension. The code generator generates a file that has the extension `.cpp`.

Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

Dependency

This parameter is enabled by **Data definition**.

Command-Line Information

Parameter: `DataDefinitionFile`

Type: `string`

Value: a valid file

Default: `'global.c'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A 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 **Code Generation** 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	A valid value
Efficiency	No impact
Safety precaution	No impact

See Also

“Data Definition and Declaration Management”

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 **Code Generation** pane: **Templates** tab.

Limitation

The code generator does not check for unique filenames. Specify filenames that do not collide with default filenames from code generation.

Dependency

This parameter is enabled by **Data declaration**.

Command-Line Information

Parameter: DataReferenceFile

Type: string

Value: a valid file

Default: 'global.h'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid file
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

Use owner from data object for data definition placement

Specify whether the model uses or ignores the ownership setting of a data object for data definition in code generation.

Settings

Default: on

On

Uses the ownership setting of the data object for data definition. This value corresponds to the `SameAsModel` value of the `ModuleNamingRule` parameter.

Off

Ignores the ownership setting of the data object for data definition. This value corresponds to the `Unspecified` value of the `ModuleNamingRule` parameter.

Command-Line Information

Parameter: EnableDataOwnership

Type: string
Value: 'on' | 'off'
Default: 'on'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	No impact

#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	A valid value
Efficiency	No impact
Safety precaution	No impact

Signal display level

Specify the persistence level for MPT signal data objects.

Settings

Default: 10

Specify an integer value indicating the persistence level for 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: a valid integer

Default: 10

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	A 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 MPT parameter data objects.

Settings

Default: 10

Specify an integer value indicating the persistence level for 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: a valid integer

Default: 10

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	A valid integer
Efficiency	No impact
Safety precaution	No impact

See Also

Selecting Persistence Level for Signals and Parameters

File packaging format

Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. You can specify a different file packaging format for each referenced model.

Settings

Default: Modular

Modular

- Outputs *model_data.c*, *model_private.h*, and *model_types.h*, in addition to generating *model.c* and *model.h*. For the contents of these files, see the table in “Generated Code Modules”.
- Supports generating separate source files for subsystems. For more information on generating code for subsystems, see “Code Generation of Subsystems”.
- If you specify **Shared code placement** as **Auto** on the **Code Generation > Interface** pane of the Configuration Parameter dialog box, some utility files are in the build directory. If you specify **Shared code placement** as **Shared location**, separate files are generated for utility code in a shared location. For more information, see “Controlling Shared Utility Code Placement”.

Compact (with separate data file)

- Conditionally outputs *model_data.c*, in addition to generating *model.c* and *model.h*.
- If you specify **Shared code placement** as **Auto** on the **Code Generation > Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *model.c*. If you specify **Shared code placement** as **Shared**

location, separate files are generated for utility code in a shared location. For more information, see “Controlling Shared Utility Code Placement”.

- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

Compact

- The contents of *model_data.c* are in *model.c*.
- The contents of *model_private.h* and *model_types.h* are in *model.h* or *model.c*.
- If you specify **Shared code placement** as **Auto** on the **Code Generation > Interface** pane of the Configuration Parameter dialog box, utility algorithms are defined in *model.c*. If you specify **Shared code placement** as **Shared location**, separate files are generated for utility code in a shared location. For more information, see “Controlling Shared Utility Code Placement”.
- Does not support separate source files for subsystems.
- Does not support models with noninlined S-functions.

Command-Line Information

Parameter: ERTFilePackagingFormat

Type: string

Value: 'Modular' | 'CompactWithDataFile' | 'Compact'

Default: 'Modular'

Recommended Settings

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

See Also

- “Customize Generated Code Modules”
- “Generate Code Modules”

- “Customize Post-Code-Generation Build Processing”

Code Generation Pane: Data Type Replacement

Replace data type names in the generated code

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	<input type="text"/>
single	real32_T	<input type="text"/>
int32	int32_T	<input type="text"/>
int16	int16_T	<input type="text"/>
int8	int8_T	<input type="text"/>
uint32	uint32_T	<input type="text"/>
uint16	uint16_T	<input type="text"/>
uint8	uint8_T	<input type="text"/>
boolean	boolean_T	<input type="text"/>
int	int_T	<input type="text"/>
uint	uint_T	<input type="text"/>
char	char_T	<input type="text"/>

In this section...

“Code Generation: Data Type Replacement Tab” on page 3-48

“Replace data type names in the generated code” on page 3-48

“Replacement Name: double” on page 3-50

“Replacement Name: single” on page 3-51

“Replacement Name: int32” on page 3-53

In this section...

“Replacement Name: int16” on page 3-54

“Replacement Name: int8” on page 3-55

“Replacement Name: uint32” on page 3-57

“Replacement Name: uint16” on page 3-58

“Replacement Name: uint8” on page 3-59

“Replacement Name: boolean” on page 3-61

“Replacement Name: int” on page 3-63

“Replacement Name: uint” on page 3-64

“Replacement Name: char” on page 3-66

Code Generation: Data Type Replacement Tab

Replace built-in data type names with user-defined replacement data type names in the generated code for your model.

Configuration

This tab is visible only if you specify an ERT-based system target file.

- 1 Select **Replace data type names in the generated code**.
- 2 In the **Replacement Name** fields, selectively specify replacement data type names to use for built-in Simulink data types.

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”

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. Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- A Simulink.NumericType object.
- The **Simulink Name** built-in data type name.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The Simulink.AliasType object has the **Data scope** parameter set to Exported.

 Off

Uses Simulink Coder names for built-in Simulink data types in generated code.

Dependencies

This parameter enables replacement for all built-in data type name in the **Data type names** table with user-defined data type names in generated code.

Command-Line Information

Parameter: EnableUserReplacementTypes

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”
- “Simulink.NumericType”

Replacement Name: double

Specify a name for `double` built-in data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, `real_T`.

Specify a string for the code generator to use as a name for `double` built-in data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `double` in the **Replacement Name** column.

To replace the **Code Generation Name** for `double` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `double`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Double`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.double`

Type: `string`

Value: The **Simulink Name**, a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.

Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	<code>''</code>

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “`Simulink.AliasType`”
- “`Simulink.NumericType`”

Replacement Name: `single`

Specify a name for `single` built-in data types in generated code.

Settings

Default: `''`

If a value is not specified, the code generator uses the **Code Generation Name**, `real32_T`.

Specify a string for the code generator to use as a name for **single** built-in data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To use the **Simulink Name**, specify `single` in the **Replacement Name** column.

To replace the **Code Generation Name** for `single` with an object:

- For a `Simulink.AliasType` object, set the `BaseType` object property to `single`.
- For a `Simulink.NumericType` object, set the `DataTypeMode` object property to `Single`.
- Specify the object name in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.single`

Type: string

Value: The **Simulink Name**, the name of a `Simulink.AliasType` object, or the name of a `Simulink.NumericType` object, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	A valid string
Efficiency	No impact
Safety precaution	' '

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”
- “Simulink.NumericType”

Replacement Name: int32

Specify names for built-in Simulink data types in generated code.

Settings

Default: ' '

If a value is not specified, the code generator uses the **Code Generation Name**, `int32_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int32_T`:

- For a `Simulink.AliasType` object, set the **BaseType** object property to `int32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int32` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: ReplacementTypes, replacementName.int32

Type: string

Value: The **Simulink Name** or the name of a Simulink.AliasType object, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	''

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Replacement Name: int16

Specify names for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, int16_T.

Specify strings for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int16_T`:

- For a `Simulink.AliasType` object, set the **BaseType** object property to `int16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.int16`

Type: string

Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.

Default: `' '`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	<code>' '</code>

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “`Simulink.AliasType`”

Replacement Name: `int8`

Specify names for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, `int8_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `int8_T`:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.
- For a `Simulink.AliasType` object, set the **BaseType** object property to `int8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `int8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.int8`

Type: `string`

Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	A valid string
Efficiency	No impact
Safety precaution	' '

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Replacement Name: uint32

Specify names for built-in Simulink data types in generated code.

Settings

Default: ' '

If a value is not specified, the code generator uses the **Code Generation Name**, `uint32_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint32_T`:

- For a `Simulink.AliasType` object, set the **BaseType** object property to `uint32`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint32 c` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: ReplacementTypes, replacementName.uint32

Type: string

Value: The **Simulink Name** or the name of a Simulink.AliasType object, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	''

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Replacement Name: uint16

Specify names for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, uint16_T.

Specify strings for the code generator to use as names for built-in Simulink data types.

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint16_T`:

- For a `Simulink.AliasType` object, set the **BaseType** object property to `uint16`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint16` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.uint16`

Type: string

Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.

Default: `' '`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	<code>' '</code>

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “`Simulink.AliasType`”

Replacement Name: `uint8`

Specify names for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, `uint8_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint8_T`:

- For a `Simulink.AliasType` object, set the **BaseType** object property to `uint8`.
- To use the built-in data type name that matches the **Code Generation Name**, specify `uint8` in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.uint8`

Type: string

Value: The **Simulink Name** or the name of a `Simulink.AliasType` object, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact

Application	Setting
Traceability	A valid string
Efficiency	No impact
Safety precaution	' '

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Replacement Name: boolean

Specify names for built-in Simulink data types in generated code.

Settings

Default: ' '

If a value is not specified, the code generator uses the **Code Generation Name**, `boolean_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

For ERT S-functions, the replacement data type can be only an 8-bit integer, `int8`, or `uint8`.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- A `Simulink.NumericType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `boolean_T`:

- For a `Simulink.AliasType` object, set the **BaseType** object property to `uint8`, `int8`, or `intn`. n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.

- For a `Simulink.NumericType` object, to replace `real_T`, set the `DataTypeMode` object property to `Boolean`. Specify the name of the `Simulink.NumericType` object in the **Replacement Name** column.
- To use the Simulink Name built-in data type name which matches the Code Generation name, specify `uint8` or `int8` or `intn`, where *n* is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**, in the **Replacement Name** column.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName`.boolean

Type: string

Value: The **Simulink Name**, a `Simulink.AliasType` object, or a `Simulink.NumericType` object, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	''

See Also

- “Replace boolean with an Integer Data Type”
- “Data Type Replacement”
- “Data Type Replacement Limitations”

- “Simulink.AliasType”
- “Simulink.NumericType”

Replacement Name: int

Specify names for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, `int_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A `Simulink.AliasType` object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `int_T`:

- For a `Simulink.AliasType` object

Set the **BaseType** object property to `intn`. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.

- To use the **Simulink Name** for `int_T`, in the **Replacement Name** column, specify `intn`.

n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: ReplacementTypes, replacementName.int

Type: string

Value: The **Simulink Name** or the name of a Simulink.AliasType, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid value
Efficiency	No impact
Safety precaution	''

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Replacement Name: uint

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, uint_T.

Specify strings for the code generator to use as names for built-in Simulink data types.

Specify the replacement name as one of the following:

- A Simulink.AliasType object.
- The **Simulink Name** built-in data type name.

To replace the **Code Generation Name** `uint_T`:

- For a `Simulink.AliasType` object

Set the **BaseType** object property to `uintn`. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column.

- To use the **Simulink Name** for `uint_T`, in the **Replacement Name** column, specify `uintn`.

n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: int**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.uint`

Type: string

Value: The **Simulink Name** or the name of a `Simulink.NumericType`, where the object exists in the base workspace.

Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	''

See Also

- “Data Type Replacement”

- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Replacement Name: char

Specify names for built-in Simulink data types in generated code.

Settings

Default: ''

If a value is not specified, the code generator uses the **Code Generation Name**, `char_T`.

Specify strings for the code generator to use as names for built-in Simulink data types.

To replace the **Code Generation Name** `char_T`, create a `Simulink.AliasType` object in the Command Window.

Set the `BaseType` object property to `intn`. Specify the name of the `Simulink.AliasType` object in the **Replacement Name** column. n is the number of bits displayed in the Configuration Parameters dialog box, **Hardware Implementation** pane > **Number of bits: char**.

An error occurs, if:

- A replacement data type specification is inconsistent with the **Simulink Name** data type.
- The `Simulink.AliasType` object has the **Data scope** parameter set to `Exported`.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes`, `replacementName.char`

Type: string

Value: The name of a `Simulink.AliasType` object, where the object exists in the base workspace.

Default: ''

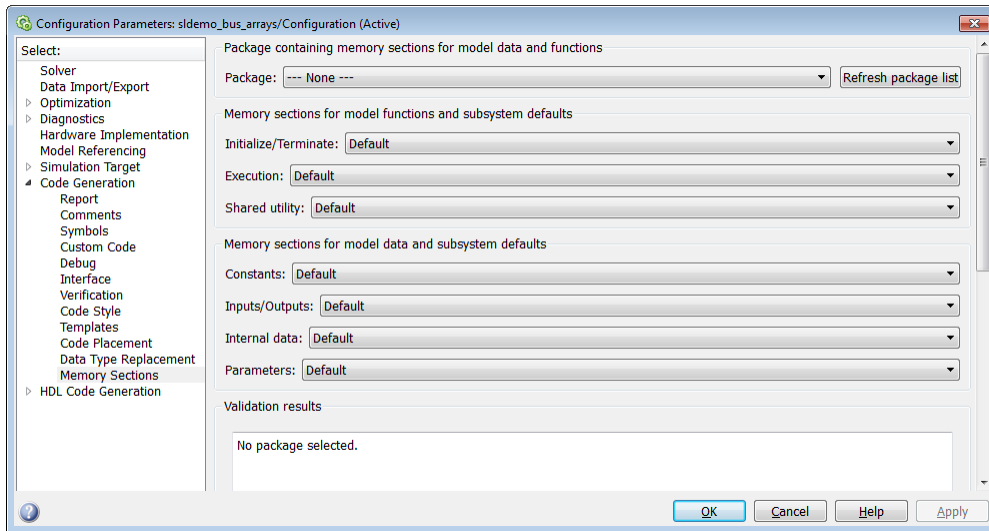
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	A valid string
Efficiency	No impact
Safety precaution	' '

See Also

- “Data Type Replacement”
- “Data Type Replacement Limitations”
- “Simulink.AliasType”

Code Generation Pane: Memory Sections



In this section...

“Code Generation: Memory Sections Tab Overview” on page 3-68

“Package” on page 3-69

“Refresh package list” on page 3-70

“Initialize/Terminate” on page 3-70

“Execution” on page 3-71

“Shared utility” on page 3-72

“Constants” on page 3-73

“Inputs/Outputs” on page 3-74

“Internal data” on page 3-75

“Parameters” on page 3-76

“Validation results” on page 3-77

Code Generation: 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”
- “Code Generation Pane: Memory Sections” on page 3-68

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 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 packages of your own, click **Refresh package list**. This action adds 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
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 packages of your own, click **Refresh package list**. This action adds 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 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 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”

Shared utility

Specify whether to apply memory sections to shared utility functions.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to subsystems except atomic subsystems that contain overriding memory section specifications.

Default: Default

Default

Suppresses the use of memory sections for shared utility functions.

memory-section-name

Applies a memory section to shared utility functions, such as fixed-point functions, lookup table functions, and binary search functions.

Command-Line Information**Parameter:** MemSecFuncSharedUtil**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 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
<i>model_cP</i>	Constant parameters

Data Definition	Data Purpose
<i>model_cB</i>	Constant block I/O
<i>model_Z</i>	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
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 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
<i>model_U</i>	Root-level input
<i>model_Y</i>	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
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 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
<i>model_B</i>	Block I/O
<i>model_D</i>	DWork vectors
<i>model_M</i>	Run-time model
<i>model_Zero</i>	Zero-crossings

Command-Line Information

Parameter: MemSecDataInternal

Type: string

Value: 'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'

Default: 'Default'

Recommended Settings

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

See Also

“Memory Sections”

Parameters

Specify whether to apply a memory section to parameters.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to 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
<i>model_P</i>	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 code generation 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

Code Generation Pane: AUTOSAR Code Generation Options

Generate XML file for schema version: 4.0

Maximum SHORT-NAME length: 32

Use AUTOSAR compiler abstraction macros

Support root-level matrix I/O using one-dimensional arrays

In this section...

“Code Generation: AUTOSAR Code Generation Options Tab Overview” on page 3-79

“Generate XML file from schema version” on page 3-79

“Maximum SHORT-NAME length” on page 3-81

“Use AUTOSAR compiler abstraction macros” on page 3-81

“Support root-level matrix I/O using one-dimensional arrays” on page 3-82

Code Generation: 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

From the Simulink **Code** menu, select **C/C++ Code > Configure Model as AUTOSAR Component** to open a dialog box where you can configure other AUTOSAR options.

See Also

- “AUTOSAR Code Generation”
- “Code Generation Pane: AUTOSAR Code Generation Options” on page 3-79

Generate XML file from schema version

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

Settings

Default: 4.0

4.1

Use schema version 4.1 (4.1.1)

4.0

Use schema version 4.0 (4.0.3)

3.2

Use schema version 3.2 (3.2.1)

3.1

Use schema version 3.1 (3.1.4)

3.0

Use schema version 3.0 (3.0.2)

2.1

Use schema version 2.1 (XSD rev 0017)

Tip

- Selecting the AUTOSAR target for your model for the first time sets the schema version parameter to the default value, 4.0.
- When you import arxml code into Simulink, the arxml importer detects the schema version and sets the schema version parameter in the model.
- From the Simulink **Code** menu, select **C/C++ Code > Configure Model as AUTOSAR Component** to open a dialog box where you can configure other AUTOSAR XML options.

Command-Line Information

Parameter: AutosarSchemaVersion

Type: string

Value: '4.1' | '4.0' | '3.2' | '3.1' | '3.0' | '2.1'

Default: '4.0'

See Also

“AUTOSAR Code Generation”

Maximum SHORT-NAME length

Specify maximum length for SHORT -NAME XML elements

Settings

Default: 32

The AUTOSAR standard specifies that the length of SHORT -NAME XML elements cannot be greater than 32 characters. This option allows you to specify a maximum length of up to 128 characters.

Command-Line Information

Parameter: AutosarMaxShortNameLength

Type: integer

Value: an integer less or equal to 128

Default: 32

See Also

“Specify Maximum SHORT-NAME Length”

Use AUTOSAR compiler abstraction macros

Specify use of AUTOSAR macros to abstract compiler directives

Settings

Default: Off



On

Software generates code with C macros that are abstracted compiler directives (near/far memory calls)



Off

Software generates code that does *not* contain AUTOSAR compiler abstraction macros.

Command-Line Information

Parameter: AutosarCompilerAbstraction

Type: string
Value: 'on' | 'off'
Default: 'off'

See Also

“Configure AUTOSAR Compiler Abstraction Macros”

Support root-level matrix I/O using one-dimensional arrays

Allow root-level matrix I/O

Settings

Default: Off

On

Software supports matrix I/O at the root-level by generating code that implements matrices as one-dimensional arrays.

Off

Software does not allow matrix I/O at the root-level. If you try to build a model that has matrix I/O at the root-level, the software produces an error.

Command-Line Information

Parameter: AutosarMatrixIOAsArray

Type: string

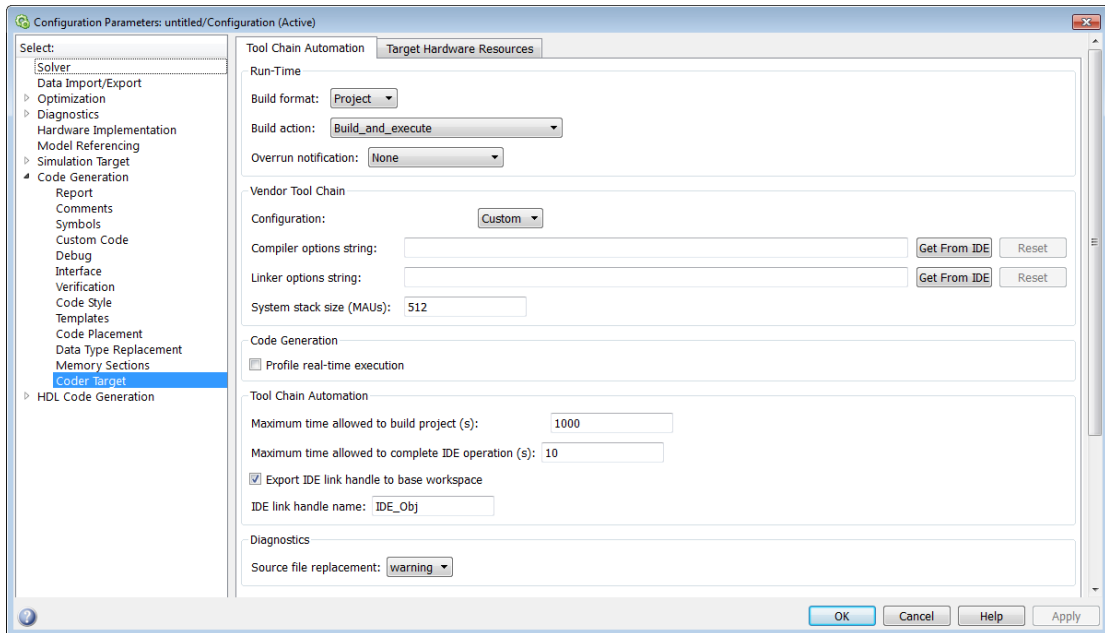
Value: 'on' | 'off'

Default: 'off'

See Also

“Root-Level Matrix I/O”

Code Generation: Coder Target Pane



In this section...

“Code Generation: Coder Target Pane Overview (previously “IDE Link Tab Overview”)” on page 3-84

“Coder Target: Tool Chain Automation Tab Overview” on page 3-84

“Build format” on page 3-86

“Build action” on page 3-87

“Overrun notification” on page 3-89

“Function name” on page 3-91

“Configuration” on page 3-91

“Compiler options string” on page 3-93

“Linker options string” on page 3-94

“System stack size (MAUs)” on page 3-95

In this section...

“System heap size (MAUs)” on page 3-96

“Profile real-time execution” on page 3-97

“Profile by” on page 3-99

“Number of profiling samples to collect” on page 3-100

“Maximum time allowed to build project (s)” on page 3-101

“Maximum time allowed to complete IDE operation (s)” on page 3-102

“Export IDE link handle to base workspace” on page 3-103

“IDE link handle name” on page 3-104

“Source file replacement” on page 3-105

Code Generation: Coder Target Pane Overview (previously “IDE Link Tab Overview”)

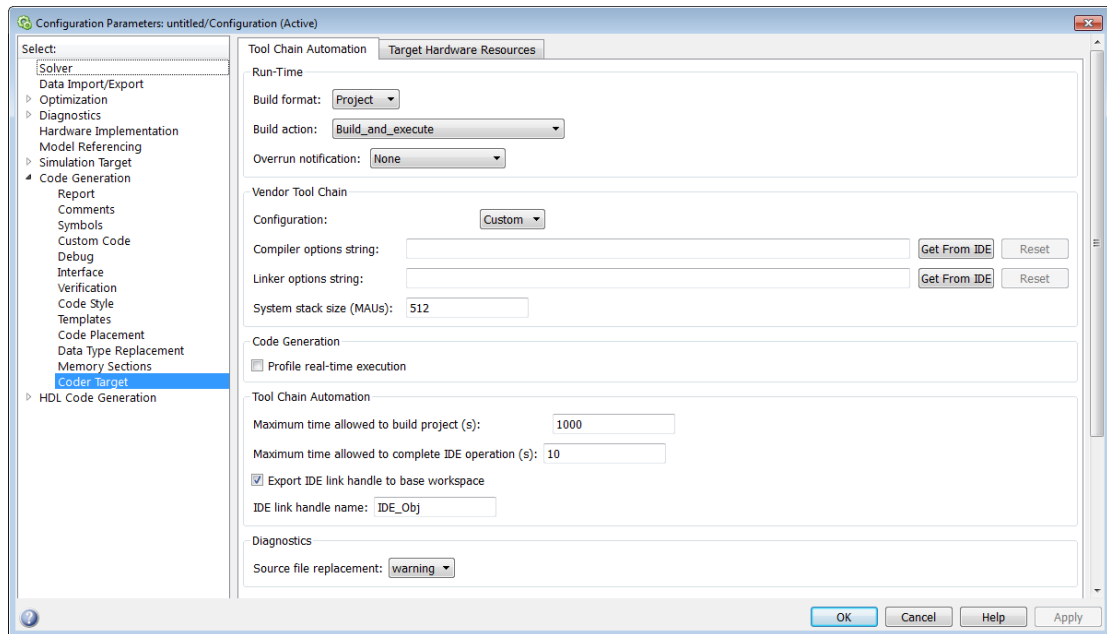
Configure the parameters for:

- Tool Chain Automation — How the coder software interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

See Also

- Coder Target: Tool Chain Automation Tab Overview
- Coder Target: Target Hardware Resources Tab Overview

Coder Target: Tool Chain Automation Tab Overview



The Tool Chain Automation Tab is only visible under the Coder Target pane.

The following table lists the parameters on the Tool Chain Automation Tab.

- “Build format” on page 3-86
- “Build action” on page 3-87
- “Overrun notification” on page 3-89
- “Function name” on page 3-91
- “Configuration” on page 3-91
- “Compiler options string” on page 3-93
- “Linker options string” on page 3-94
- “System stack size (MAUs)” on page 3-95
- “System heap size (MAUs)” on page 3-96
- “Profile real-time execution” on page 3-97
- “Profile by” on page 3-99
- “Number of profiling samples to collect” on page 3-100

- “Maximum time allowed to build project (s)” on page 3-101
- “Maximum time allowed to complete IDE operation (s)” on page 3-102
- “Export IDE link handle to base workspace” on page 3-103
- “IDE link handle name” on page 3-104
- “Source file replacement” on page 3-105

Build format

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Project

Project

Builds your model as an IDE project.

Makefile

Creates a makefile and uses it to build your model.

Dependencies

Selecting **Makefile** removes the following parameters:

- **Code Generation**
 - **Profile real-time execution**
 - **Profile by**
 - **Number of profiling samples to collect**
- **Link Automation**
 - **Maximum time allowed to build project (s)**
 - **Maximum time allowed to complete IDE operation (s)**
 - **Export IDE link handle to base workspace**
 - **IDE link handle name**

Command-Line Information

Parameter: buildFormat

Type: string

Value: Project | Makefile

Default: Build_and_execute

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Build_and_execute

If you set **Build format** to **Project**, select one of the following options:

Build_and_execute

Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

Create_project

Directs Simulink Coder software to create a new project in the IDE. The command line equivalent for this setting is **Create**.

Archive_library

Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

Build

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

Create_processor_in_the_loop_project

Directs the Simulink Coder code generation process to create PIL algorithm object code as part of the project build.

If you set **Build format** to **Makefile**, select one of the following options:

Create_makefile

Creates a makefile. For example, “.mk”. The command line equivalent for this setting is **Create**.

Archive_library

Creates a makefile and an archive library. For example, “.a” or “.lib”.

Build

Creates a makefile and an executable. For example, “.exe”.

Build_and_execute

Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

Dependencies

Selecting **Archive_library** removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting **Create_processor_in_the_loop_project** removes the following parameters:

- **Overrun notification**

- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: buildAction

Type: string

Value: Build | Build_and_execute | Create | Archive_library | Create_processor_in_the_loop_project

Default: Build_and_execute

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

Overrun notification

Specifies how your program responds to overrun conditions during execution.

Settings

Default: None

None

Your program does not notify you when it encounters an overrun condition.

Print_message

Your program prints a message to standard output when it encounters an overrun condition.

Call_custom_function

When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting `Call_custom_function` enables the **Function name** parameter.

Setting this parameter to `Call_custom_function` enables the **Function name** parameter.

Command-Line Information

Parameter: `overrunNotificationMethod`

Type: string

Value: `None` | `Print_message` | `Call_custom_function`

Default: `None`

Recommended Settings

Application	Setting
Debugging	<code>Print_message</code> or <code>Call_custom_function</code>
Traceability	<code>Print_message</code>
Efficiency	<code>None</code>
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

Command-Line Information

Parameter: `overrunNotificationFcn`

Type: string

Value: no default

Default: no default

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Configuration

Sets the Configuration for building your project from the model.

Settings

Default: Custom

Custom

Lets the user apply a specialized combination of build and optimization settings.

Custom applies the same settings as the Release project configuration in IDE, except:

- The compiler options do not use optimizations.
- The memory configuration specifies a memory model that uses **Far Aggregate** for data and **Far** for functions.

Debug

Applies the **Debug** Configuration defined by the IDE to the generated project and code.

Release

Applies the **Release** project configuration defined by the IDE to the generated project and code.

Dependencies

- Selecting **Custom** disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting **Release** sets the **Compiler options string** to the settings defined by the IDE.
- Selecting **Debug** sets the **Compiler options string** to the settings defined by the IDE.

Command-Line Information

Parameter: projectOptions

Type: string

Value: Custom | Debug | Release

Default: Custom

Recommended Settings

Application	Setting
Debugging	Custom or Debug
Traceability	Custom, Debug, Release

Application	Setting
Efficiency	Release
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the coder product does not set optimization flags.

With Texas Instruments Code Composer Studio v3.3 and Analog Devices VisualDSP++, the user interface displays **Get From IDE** and **Reset** buttons next to this parameter. If you have an active project open in the IDE, you can click **Get From IDE** to import the compiler option setting from the current project in the IDE. To reset the compiler option to the default value, click **Reset**.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Configuration** to **Custom** applies the **Custom** compiler options defined by coder software. **Custom** does not use optimizations.
- Setting **Configuration** to **Debug** applies the debug settings defined by the IDE.
- Setting **Configuration** to **Release** applies the release settings defined by the IDE.

Command-Line Information

Parameter: compilerOptionsStr

Type: string

Value: Custom | Debug | Release

Default: Custom

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the coder product does not set linker options.

With Texas Instruments Code Composer Studio v3.3 and Analog Devices VisualDSP++, the user interface displays **Get From IDE** and **Reset** buttons next to this parameter. If you have an active project open in the IDE, you can click **Get From IDE** to import the linker options string from the current project in the IDE. To reset the linker options to the default value of no options, click **Reset**.

Settings

Default: No default

Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: `linkerOptionsStr`

Type: `string`

Value: valid linker option

Default: none

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory.

This parameter is used in targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For operating systems such as Linux or VxWorks, this value specifies the stack space allocated per thread.

This parameter also affects the “Maximum stack size (bytes)” parameter, located in the Optimization > Signals and Parameters pane.

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the stack size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.

- The software does not verify the value you entered is valid.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

When you set the **System target file** parameter on the **Code Generation** pane to `idelink_ert.tlc` or `idelink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization > Signals and Parameters** pane to `Inherit from target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of $(\text{System stack size}/2)$ with 200,000 bytes and uses the smaller of the two values.

Command-Line Information

Parameter: `systemStackSize`

Type: `int`

Default: 8192

Recommended Settings

Application	Setting
Debugging	<code>int</code>
Traceability	<code>int</code>
Efficiency	<code>int</code>
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

System heap size (MAUs)

Set the default heap size that the target processor reserves for dynamic memory allocation.

The target processor uses this heap for functions like `printf()` and system services code.

The following IDEs use this parameter:

- Analog Devices VisualDSP++

- Wind River Diab/GCC (makefile generation only)

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the heap size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify that your size is valid. Be sure that you enter an acceptable value.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

Command-Line Information

Parameter: `systemHeapSize`

Type: int

Default: 8192

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

Settings

Default: Off

On

Adds instrumentation to the generated code to support execution profiling and generate the profiling report.

Off

Does not instrument the generated code to produce the profile report.

Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop` project removes this parameter.

Command-Line Information

Parameter: ProfileGenCode

Type: string

Value: 'on' | 'off'

Default: 'off'

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics..

Profile by

Defines which execution profiling technique to use.

Settings

Default: Task

Task

Profiles model execution by the tasks in the model.

Atomic subsystem

Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information

Parameter: profileBy

Type: string

Value: Task | Atomic subsystem

Default: Task

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics.

Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

Settings

Default: 100

Minimum: 2

Maximum: Buffer capacity

Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information

Parameter: ProfileNumSamples

Type: int

Value: Positive integer

Default: 100

Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

Command-Line Information

Parameter: `ideObjBuildTimeout`

Type: int

Value: Integer greater than 0

Default: 100

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as `read` or `write`, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information

Parameter: 'ideObjTimeout'

Type: int

Value:
Default: 10

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Export IDE link handle to base workspace

Directs the software to export the IDE_Obj object to your MATLAB workspace.

Settings

Default: On

On

Directs the build process to export the IDE_Obj object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.

Off

prevents the build process from exporting the IDE_Obj object to your MATLAB software workspace.

Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

Command-Line Information

Parameter: exportIDEObj

Type: string

Value: 'on' | 'off'

Default: 'on'

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

IDE link handle name

specifies the name of the IDE_Obj object that the build process creates.

Settings

Default: IDE_Obj

- Enter a valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the IDE_Obj object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information

Parameter: ideObjName

Type: string

Value:

Default: IDE_Obj

Recommended Settings

Application	Setting
Debugging	Enter a valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Source file replacement

Selects the diagnostic action to take if the coder software detects conflicts that you are replacing source code with custom code.

Settings

Default: warn

none

Does not generate warnings or errors when it finds conflicts.

warning

Displays a warning.

error

Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select **warning** and the software detects custom code replacement. You see warning messages as the build progresses.

- Select **error** the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select **none** when you do not want to see multiple messages during your build.
- The messages apply to Simulink Coder **Custom Code** replacement options as well.

Command-Line Information

Parameter: DiagnosticActions

Type: string

Value: none | warning | error

Default: warning

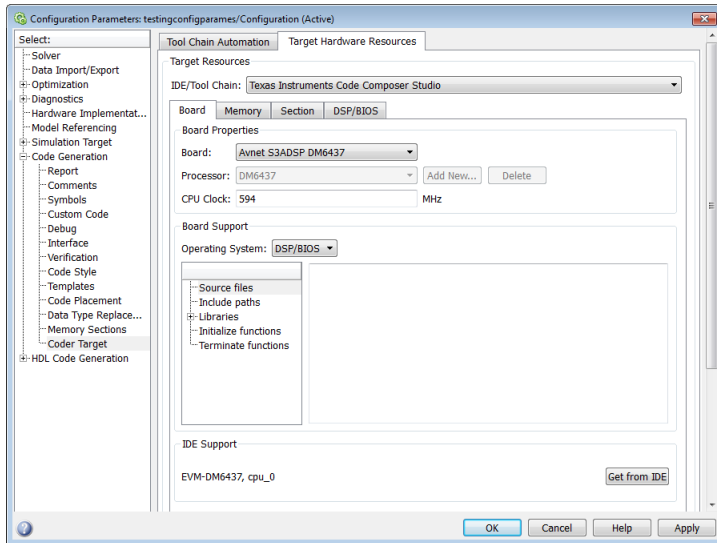
Recommended Settings

Application	Setting
Debugging	error
Traceability	error
Efficiency	warning
Safety precaution	error

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

Code Generation: Target Hardware Resources Pane



In this section...

- “Code Generation: Coder Target Pane Overview” on page 3-108
- “(Target Hardware Resources)” on page 3-108
- “Coder Target: Target Hardware Resources Tab Overview” on page 3-108
- “IDE/Tool Chain” on page 3-109
- “Target Hardware Resources: Board Tab” on page 3-110
- “Target Hardware Resources: Memory Tab” on page 3-113
- “Target Hardware Resources: Section Tab” on page 3-115
- “Target Hardware Resources: DSP/BIOS Tab” on page 3-118
- “Target Hardware Resources: Peripherals Tab” on page 3-121
- “Clocking” on page 3-123
- “ADC” on page 3-126
- “COMP” on page 3-128
- “eCAN_A, eCAN_B” on page 3-129

In this section...

“eCAP” on page 3-131

“ePWM” on page 3-133

“I2C” on page 3-135

“SCI_A, SCI_B, SCI_C” on page 3-141

“SPI_A, SPI_B, SPI_C, SPI_D” on page 3-144

“eQEP” on page 3-147

“Watchdog” on page 3-149

“GPIO” on page 3-151

“Flash_loader” on page 3-156

“DMA_ch[#]” on page 3-158

“LIN” on page 3-167

“Add Processor Dialog Box” on page 3-174

“Target Hardware Resources Tab: Linux, VxWorks, or Windows” on page 3-175

Code Generation: Coder Target Pane Overview

Control options for third-party software build toolchains and processors.

Configuration

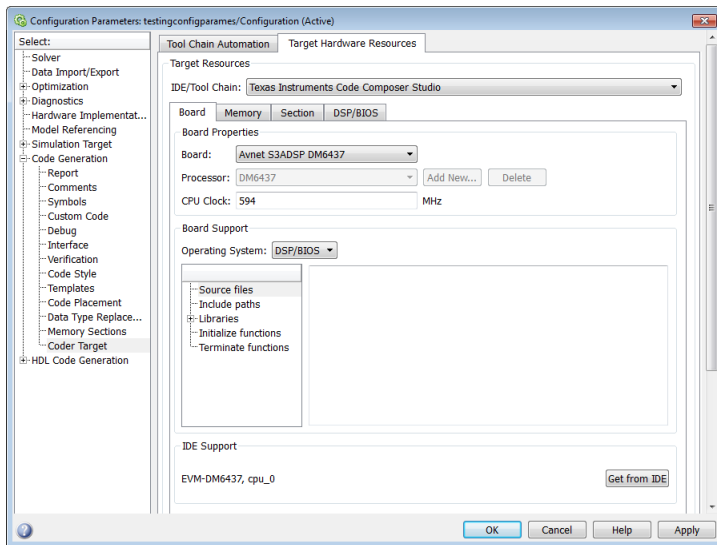
This tab appears only if you specify an `idelink_ert` or `idelink_grt` system target file.

(Target Hardware Resources)

Configure the parameters for:

- Tool Chain Automation — How the coder software interacts with third-party software build toolchains.
- Target Hardware Resources — The IDE toolchain and properties of the physical hardware, such as board, operating system, memory, and peripherals.

Coder Target: Target Hardware Resources Tab Overview



The Target Hardware Resources tab is only visible under the Coder Target pane.

The following table lists the parameters and tabs on the Target Hardware Resources tab.

- “IDE/Tool Chain” on page 3-109
- “Target Hardware Resources: Board Tab” on page 3-110
- “Target Hardware Resources: Memory Tab” on page 3-113
- “Target Hardware Resources: Section Tab” on page 3-115
- “Target Hardware Resources: Peripherals Tab” on page 3-121

IDE/Tool Chain

Select the IDE or software build tool chain you are using from the list of options. This action applies parameter values for a specific IDE or tool chain.

Located on the Target Hardware Resources tab.

Settings

The name of a specific toolchain

The name of a specific toolchain appears after you install an Embedded Coder support package.

Selecting the toolchain configures the Target Hardware Resources parameters to work with a specific toolchain.

To install a support package, select `Get more...` or enter `supportPackageInstaller` in the MATLAB Command Window.

`Get more...`

Launches the Support Package Installer. For more information, see `supportPackageInstaller`.

See Also

`supportPackageInstaller`

Target Hardware Resources: Board Tab

The following options appear on the **Board** pane, which has separate panels for **Board Properties**, **Board Support**, and **IDE Support** labels.

Board

Select your target board from the list of options. Selecting a specific board sets the value for the **Processor** parameter. If you select a custom board, also set the **Processor** parameter.

Processor

The Board and Processor settings apply default values to many of the parameters, such as those under the **Memory** and **Section** tabs.

If the coder product supports an operating system for the processor, it enables the **Operating system** option.

Note Selecting or reselecting a processor resets the solver and some processor-specific parameters to their default values.

Add New

Clicking **Add new** opens a new dialog box where you specify configuration information for a processor that is not on the Processor list.

For details about the New Processor dialog box, refer to “Add Processor Dialog Box” on page 3-174.

Delete

Clicking **Delete**, removes a processor that you added to the **Processor** list. You cannot delete the standard processors.

CPU Clock

Enter the actual clock rate the board uses. This action does not change the rate on the board. Rather, the code generation process requires this information to produce code that runs on the hardware. Setting this value incorrectly causes timing and profiling errors when you run the code on the hardware.

The timer uses the value of **CPU clock** to calculate the time for each interrupt. For example, a model with a sine wave generator block running at 1 kHz uses timer interrupts to generate sine wave samples at the specified rate. For example, using 100 MHz, the timer calculates the sine generator interrupt period as follows:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

$100,000,000/1000 = 1$ Sine block interrupt per 100,000 clock ticks

Board Support

Select the following parameters and edit their values in the text box on the right:

- **Source files** — Enter the full paths to source code files.
- **Include paths** — Add paths to include files.
- **Libraries** — Identify specific libraries for the processor. Required libraries appear on the list by default. To add more libraries, entering the full path to the library with the library file in the text area.
- **Initialize functions** — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.
- **Terminate functions** — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

Note Invalid or incorrect entries in these fields can cause errors during code generation. When you enter a file path, library, or function, the block does not verify that the path or function exists or is valid.

When entering a path to a file, library, or other custom code, use the following string in the path to refer to the IDE installation folder.

```
$(Install_dir)
```

Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code. **Support** options do not support functions that use return arguments or values. These parameters accept only functions of type `void fname void` as valid as entries.

You can also set up environment variables to use as folder path tokens. For example, if you set up an environment called `USER_VAR`, you can use it as a token when you define a path in Coder Target > Target Hardware Resources. For example:
`$(USER_VAR)\myinstal\foo.c.`

Operating System

Select an operating system or RTOS for your target. If your target platform supports an operating system, the software enables the **Operating system** parameter. Otherwise, the software disables this option.

Get from IDE

This button only appears when you are using Texas Instruments Code Composer Studio 3.3 IDE or Analog Devices VisualDSP++ IDE:

- With Texas Instruments Code Composer Studio 3.3 IDE, the **Get from IDE** button imports the current **Board Name** and **Processor Name** from the IDE.
- With Analog Devices VisualDSP++ IDE, the **Get from IDE** button imports the current **Session Name** and **Processor Name** from the IDE.

Use the **Get from IDE** button to update the Coder Target > Target Hardware Resources, the IDE, and the hardware board so they refer to the same processor. Otherwise, during code generation, the software generates a warning similar to the following message:

```
Target Hardware Resources tab specifies that the board named
'<boardname1>' will be used to run generated code.
However, since only board named '<boardname2>' is found
```


in your system, that board will be used.

Board Name

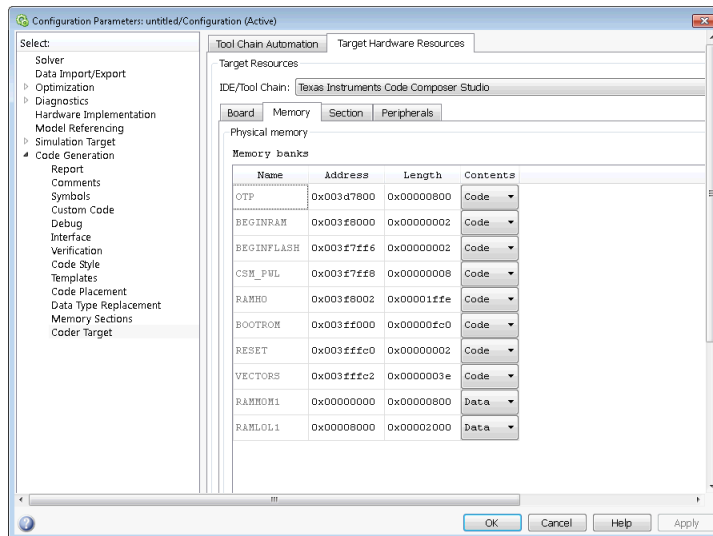
Board Name appears after you click **Get from IDE**. Select the board you are using. Match **Board Name** with the **Board** option near the top of the **Board** pane.

Processor Name

Processor Name appears after you click **Get from IDE**. If the board you selected in **Board Name** has multiple processors, select the processor you are using. Match **Processor Name** with the **Processor** option near the top of the **Board** pane.

Note: Click **Apply** to update the board and processor description under **IDE Support**.

Target Hardware Resources: Memory Tab



After selecting a board, specify the layout of the physical memory on your processor and board to determine how to use it for your program.

The **Memory** pane contains memory options for:

- **Physical Memory** — Specifies the processor and board memory map

- **Cache Configuration** — Select a cache configuration where available, such as L2 cache, and select one of the corresponding configuration options, such as 32 kb.

For more information about memory segments and memory allocation, consult the reference documentation for the IDE or processor.

The **Physical Memory** table shows the memory segments or *memory banks* available on the board and processor. By default, Target Hardware Resources tab show the memory segments found on the selected processor. In addition, the **Memory** pane on Target Hardware Resources tab shows the memory segments available on the board, but external to the processor. Target Hardware Resources tab set default starting addresses, lengths, and contents of the default memory segments. The default memory segments for each processor and board differ.

Click **Add** to add physical memory segments to the **Memory banks** table.

After you add the segment, you can configure the starting address, length, and contents for the new segment.

Name

To change the memory segment name, click the name, and then type the new name. Names are case sensitive. **NewSegment** is not the same as **newsegment** or **newSegment**.

Note You cannot rename default processor memory segments (name in gray text).

Address

Address reports the starting address for the memory segment showing in **Name**. Address entries appear in hexadecimal format and are limited only by the board or processor memory.

Length

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs).

For the C6000 processor family, the MADU requires inputs of 8 bytes, one word.

Contents

Configure the segment to store **Code**, **Data**, or **Code & Data**. Changing processors changes the options for each segment.

You can add and use as many segments of each type as you need, within the limits of the memory on your processor. Every processor must have a segment that holds code, and a segment that holds data.

Add

Click **Add** to add a new memory segment to the processor memory map. When you click **Add**, a new segment name appears, for example **NEWMEM1**, in **Name** and on the **Memory banks** table. In **Name**, change the temporary name **NEWMEM1** by entering the new segment name. Entering the new name, or clicking **Apply**, updates the temporary name on the table to the name you enter.

Remove

This option lets you remove a memory segment from the memory map. Select the segment to remove on the **Memory banks** table, and click **Remove** to delete the segment.

Cache (Configuration)

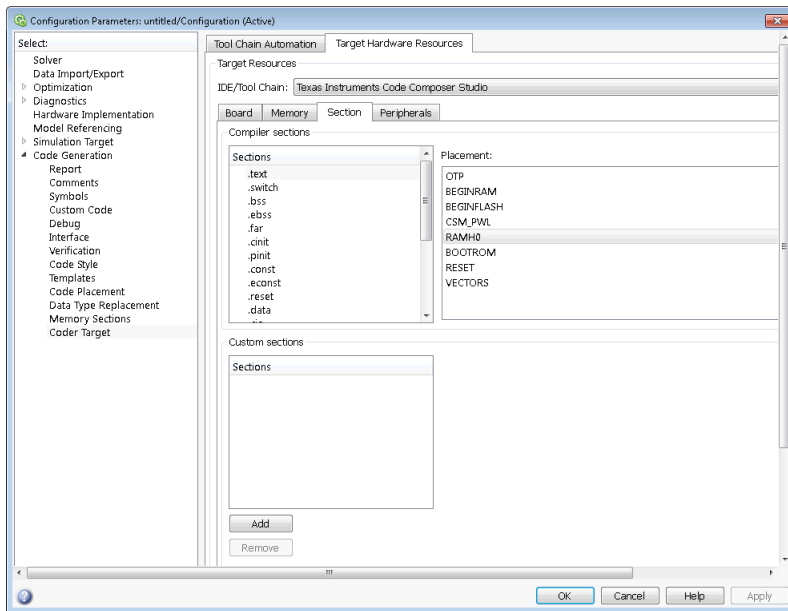
When the **Processor** on the Board pane supports a cache memory structure, the dialog box displays a table of **Cache** parameters. You can use this table to configure the cache as SRAM and partial cache. Both the data memory and the program share this second-level memory.

If your processor supports the two-level memory scheme, this option enables the L2 cache on the processor.

Some processors support code base memory organization. For example, you can configure part of internal memory as code.

Cache level lets you select one of the available cache levels to configure by selecting one of its configurations. For example, you can select L2 cache level, and choose one of its configurations, such as 32 kb.

Target Hardware Resources: Section Tab



Options on this pane specify where program sections appear in memory. Program sections differ from memory segments—sections comprise portions of the executable code stored in contiguous memory locations. Commonly used sections include `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler, and some can be custom sections.

For more information about program sections and objects, refer to the online help for your IDE.

Within the Section pane, you configure the allocation of sections for **Compiler** and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections** and **Custom sections** lists in the pane. All sections do not appear on all lists.

String	Section List	Description of the Section Contents
<code>.bss</code>	Compiler	Static and global C variables in the code
<code>.cinit</code>	Compiler	Tables for initializing global and static variables and constants

String	Section List	Description of the Section Contents
.cio	Compiler	Standard I/O buffer for C programs
.const	Compiler	Data defined with the C qualifier and string constants
.data	Compiler	Program data for execution
.far	Compiler	Variables, both static and global, defined as far variables
.pinit	Compiler	Load allocation of the table of global object constructors section
.stack	Compiler	The global stack
.switch	Compiler	Jump tables for switch statements in the executable code
.system	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants

You can learn more about memory sections and objects in the online help for your IDE.

Default Sections

When you highlight a section on the list, **Description** show a brief description of the section. Also, **Placement** shows you the memory allocation of the section.

Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

Placement

Shows the allocation of the selected **Compiler sections** entry in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments as defined in the physical memory map on the **Memory** pane. Select one of the listed memory segments to allocate the highlighted compiler section to the segment.

To see a description of the placement item, hover your mouse pointer over the item for a few moments.

Custom Sections

If your program uses code or data sections that are not in the **Compiler sections**, add the new sections to **Custom sections**.

Sections

This window lists data sections that are not in the **Compiler sections**.

Placement

With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select a segment that appears on the list.

Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. You can also click **OK** to add the section to the list and close the dialog box.

Name

Enter the name of the new section here. To add a new section, click **Add**. Then, replace the temporary name with the name to use. Although the temporary name includes a period at the beginning you do not need to include the period in your new name. Names are case sensitive. **NewSection** is not the same as **newsection**, or **newSection**.

Contents

Identify whether the contents of the new section are **Code**, **Data**, or **Any**.

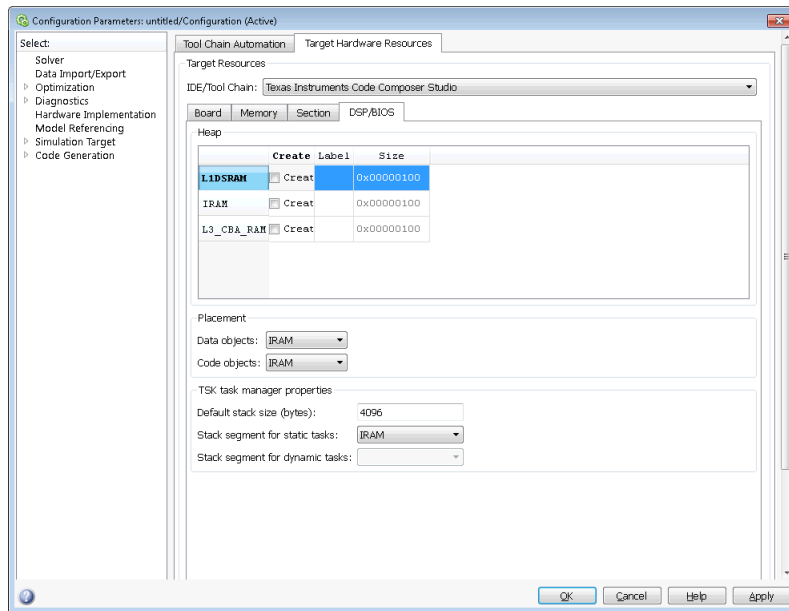
Remove

To remove a section from the **Custom sections** list, select the section and click **Remove**.

Target Hardware Resources: DSP/BIOS Tab

The DSP/BIOS pane is available if the two following conditions are true:

- You are using Texas Instruments CCS IDE.
- You set the Target Hardware Resources tab **Processor** option to a C6000 processor that supports DSP/BIOS.



Selecting DSP/BIOS for **Operating system** on the Board pane enables this pane.

Use the **Heap**, **Placement**, and **TSK task manager properties** sections of this pane to configure various modules of DSP/BIOS.

For more information about tasks, refer to the Code Composer Studio online help.

Note To enable the **Heap** option, select DSP/BIOS for **Operating system** on the **Board** pane.

Heap

The heap section contains the **Create**, **Label**, and **Size** options to manage the heap.

Create

If your processor supports using a heap, selecting this option enables creating the heap. Define the heap using the **Label** and **Size** options. **Create** becomes unavailable for processors that do not provide a heap or do not allow you to configure the heap.

The location of the heap in the memory segment is not under your control. The only way to control the location of the heap in a segment is to make the segment and the heap the same size. Otherwise, the compiler determines the location of the heap in the segment.

Size

After you select **Create**, this option lets you specify the size of the heap in words. Enter the number of words in decimal format. When you enter the heap size in decimal words, the system converts the decimal value to hexadecimal format. You can enter the value directly in hexadecimal format as well. Processors can support different maximum heap sizes.

Label

Selecting **Create** enables this option. Enter your label for the heap in the **Heap** option.

Note When you enter a label, the block does not verify that the label is valid. An invalid label in this field can cause errors during code generation.

Placement

Use the **Data object** and **Code object** options in **Placement** to configure the memory allocation of the selected **Heap** list entry.

Data object

Specify where to place new data objects in memory.

Code object

Specify where to place new code objects in memory.

TSK task manager properties

Use the **Default stack size (bytes)**, **Stack segment for static tasks**, and **Stack segment for dynamic tasks** options in **TSK task manager properties** to configure the task manager properties.

Default stack size (bytes)

DSP/BIOS uses a stack to save and restore variables and CPU context during thread preemption for task threads. This option sets the size of the DSP/BIOS stack in bytes allocated for each task. The software sets the default value to 4096 bytes. The maximum value is determined by the processor. Set the stack size so that tasks do

not use more memory than you allocate. Exceeding the stack memory size can cause the task to write into other memory or data areas, causing unpredictable behavior.

Stack segment for static tasks

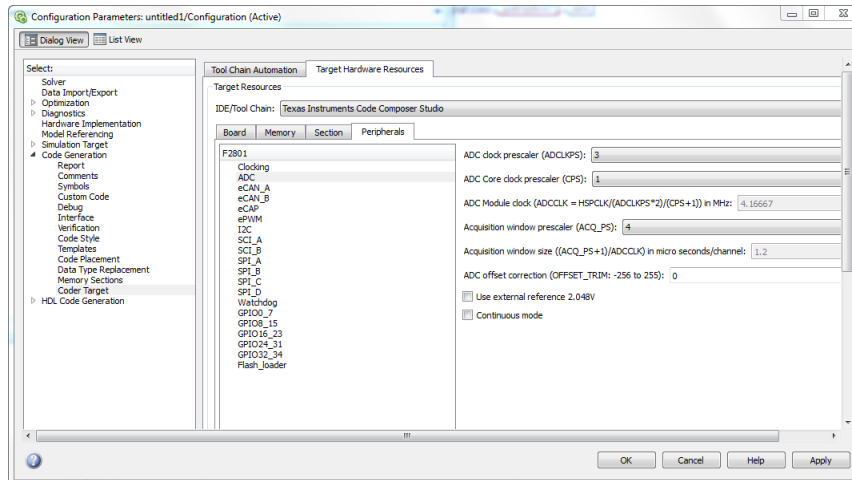
Use this option to specify where to allocate the stack for static tasks. Tasks that your program uses often are good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

The list offers IDRAM for locating the stack in memory. The Memory pane provides more options for the physical memory on the processor.

Stack segment for dynamic tasks

Like static tasks, dynamic tasks use a stack as well. Setting this option specifies where to locate the stack for dynamic tasks. In this case, MEM_NULL is the only valid stack location in memory. Allocate system heap storage to use this option. Specify the system heap configuration on the “Target Hardware Resources: Memory Tab” on page 3-113.

Target Hardware Resources: Peripherals Tab



The Peripherals pane is only visible under the Target Hardware Resources tab, when **Board** and **Processor** parameters are configured for a C2000 processors.

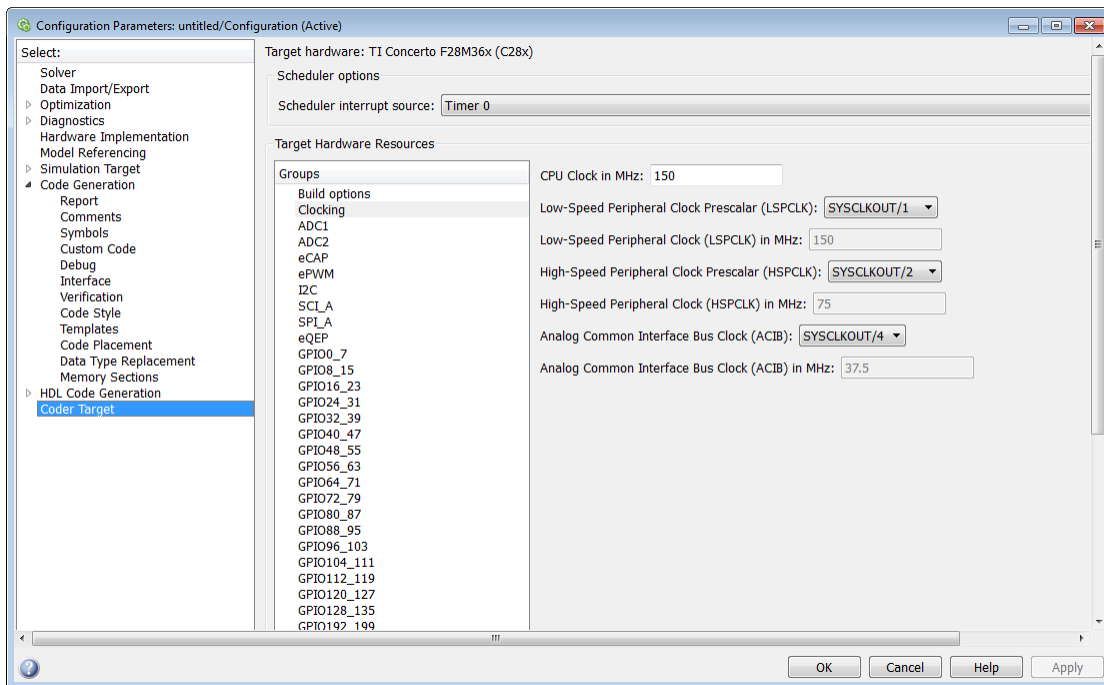
To set the attributes for a peripheral, select the peripheral from the **Peripherals** list and then set the attribute options on the right side.

The following table describes all the peripherals provided on the **Peripherals** list. Some peripherals are not available on some C2000 processors.

Peripheral Name	Description
“Clocking” on page 3-123	Clocking parameters to adjust clock settings and match custom oscillator frequencies
“ADC” on page 3-126	Analog-to-Digital Converter (ADC) parameters
“COMP” on page 3-128	Parameters to assign COMP pins to GPIO pins.
“eCAN_A, eCAN_B” on page 3-129	Enhanced Controller Area Network (eCAN) parameters for modules A or B
“eCAP” on page 3-131	Enhanced Capture (eCAP) parameters for pin mapping to GPIO
“ePWM” on page 3-133	Enhanced Pulse Width Modulation (ePWM) parameters for pin mapping to GPIO
“I2C” on page 3-135	Inter-Integrated Circuit (I2C) parameters for communications
“SCI_A, SCI_B, SCI_C” on page 3-141	Serial Communications Interface (SCI) parameters for communications with modules A, B, or C
“SPI_A, SPI_B, SPI_C, SPI_D” on page 3-144	Serial Peripheral Interface (SPI) parameters for communications with module A, B, C, or D
“eQEP” on page 3-147	Enhanced Quadrature Encoder Pulse (eQEP) parameters for pin mapping to GPIO
“Watchdog” on page 3-149	Watchdog enable/disable and timing
“GPIO” on page 3-151	General Purpose Input Output (GPIO) parameters for input qualification types
“Flash_loader” on page 3-156	Flash memory loader/programmer
“DMA_ch[#]” on page 3-158	Direct Memory Access (DMA) parameters for channels 1 to N

Peripheral Name	Description
“LIN” on page 3-167	Local Interconnect Network (LIN) parameters for communications

Clocking



Use the clocking options to help you achieve the CPU Clock rate specified on the board. The default clocking values run the CPU clock (CLKIN) at its maximum frequency. The parameters use the external oscillator frequency on the board (OSCCLK) that is recommended by the processor vendor.

You can get feedback on the closest achievable SYSCLKOUT value with the specified Oscillator clock frequency by selecting the **Auto set PLL based on OSCCLK and CPU clock** check box. Alternatively, you can manually specify the PLL value for the SYSCLKOUT value calculation.

Change the clocking values if:

- You want to change the CPU frequency.
- The external oscillator frequency differs from the value recommended by the manufacturer.

To determine the CPU frequency (CLKIN), use the following equation:

$$\text{CLKIN} = (\text{OSCCLK} * \text{PLLCR}) / (\text{DIVSEL or CLKINDIV})$$

- CLKIN is the frequency at which the CPU operates, also known as the CPU clock.
- OSCCLK is the frequency of the oscillator.
- **PLLCR** is the PLL Control Register value.
- **CLKINDIV** is the Clock in Divider.
- **DIVSEL** is the Divider Select.

The availability of the DIVSEL or CLKINDIV parameters changes depending on the processor that you select. If neither parameter is available, use the following equation:

$$\text{CLKIN} = (\text{OSCCLK} * \text{PLLCR}) / 2$$

In the **CPU clock**

parameter of the Coder Target > Target Hardware Resources tab, enter the resulting CPU clock frequency (CLKIN).

For more information, see the “PLL-Based Clock Module” section in the Texas Instruments *Reference Guide* for your processor.

Use internal oscillator

Use the internal zero pin oscillator on the CPU. This parameter is enabled by default.

Oscillator clock (OSCCLK) frequency in MHz

The oscillator frequency that is used in the processor. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Auto set PLL based on OSCCLK and CPU clock

The option that helps you set the PLL control register value automatically. When you select this check box, the values in the PLLCR, DIVSEL, and the Closest achievable SYSCLKOUT in MHz parameters are automatically calculated based on the **CPU**

Clock value entered on the Board. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

PLL control register (PLLCR)

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated control register value achieves the specified CPU Clock value, based on the Oscillator clock frequency. Otherwise, you can select a value for PLL control register. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Clock divider (DIVSEL)

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (DIVSEL). This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/DIVSEL **Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/CLKINDIV**

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, PLLCR, and the DIVSEL. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Low-Speed Peripheral Clock Prescaler (LSPCLK)

The value by which to scale the LSPCLK. This value is based on the SYSCLKOUT.

Low-Speed Peripheral Clock (LSPCLK) in MHz

This value is calculated based on LSPCLK Prescaler. Example: SPI uses a LSPCLK.

High-Speed Peripheral Clock Prescaler (HSPCLK)

The value by which to scale the HSPCLK. This value is based on the SYSCLKOUT.

High-Speed Peripheral Clock (HSPCLK) in MHz

This value is calculated based on HSPCLK Prescaler. Example: ADC uses a HSPCLK.

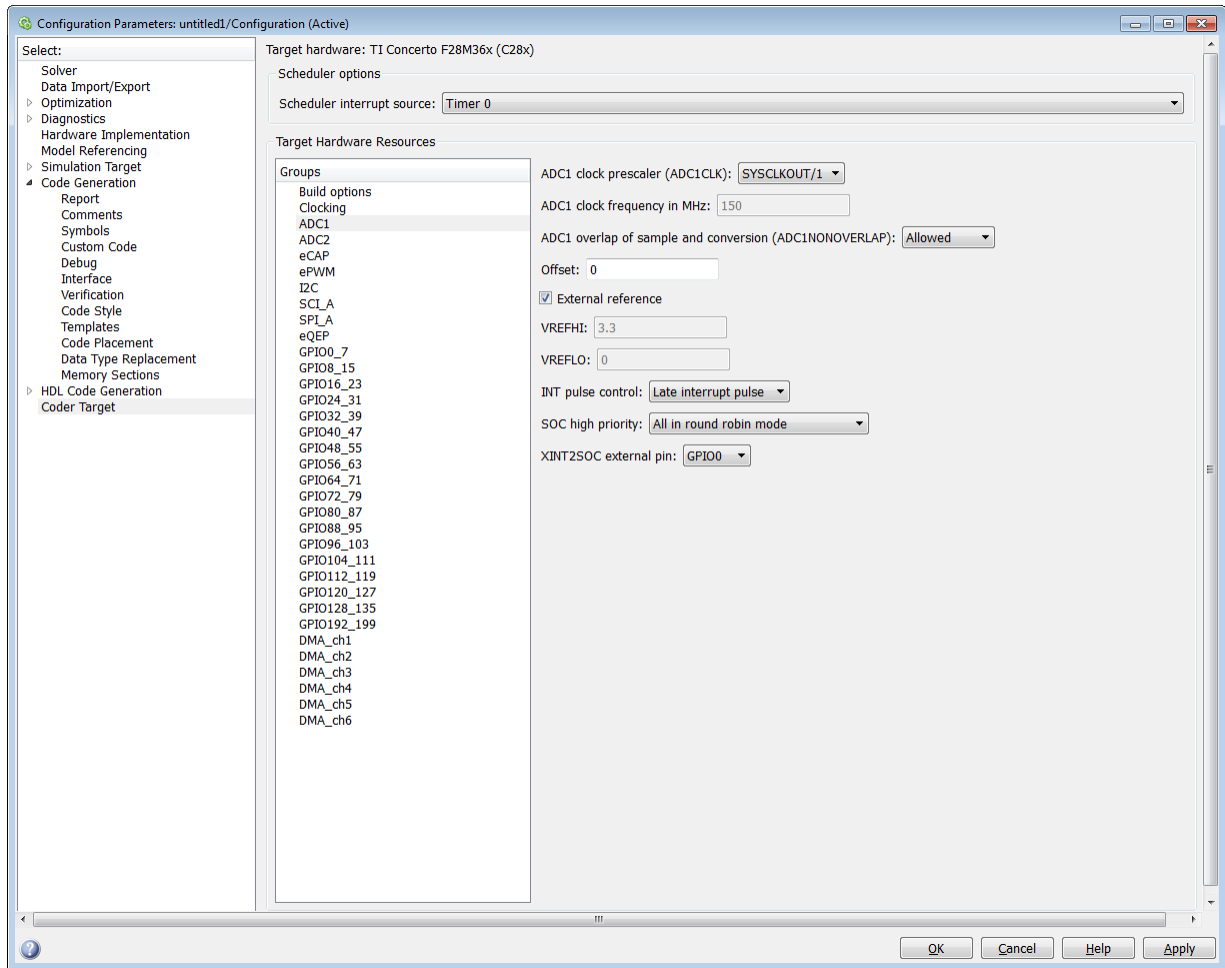
Analog Common Interface Bus Clock (ACIB)

The value by which to scale the bus clock. This option is available only for TI Concerto F28M35x/ F28M36x processors.

Analog Common Interface Bus Clock (ACIB) in MHz

This value is calculated based on the ACIB value. This option is available only for TI Concerto F28M35x/ F28M36x processors.

ADC



The high-speed peripheral clock (HSPCLK) controls the internal timing of the ADC module. The ADC derives the operating clock speed from the HSPCLK speed in several prescaler stages. For more information about configuring these scalars, refer to “Configuring ADC Parameters for Acquisition Window Width”.

You can set the following parameters for the ADC clock prescaler:

ADC clock prescaler (ADCCLK)

The option to select the ADCCLK divider for processors c2802x, c2803x, c2806x, or F28M3x.

ADC clock frequency in MHz

The clock frequency for ADC. This is a read-only field and the value in this field is based on the value you select in **ADC clock prescaler (ADCCLK)**.

ADC overlap of sample and conversion (ADC#NONOVERLAP)

The option to enable or disable overlap of sample and conversion.

ADC clock prescaler (ADCLKPS)

The HSPCLK speed is divided by this 4-bit value as the first step in deriving the core clock speed of the ADC. The default value is 3.

ADC Core clock prescaler (CPS)

After dividing the HSPCLK speed by the **ADC clock prescaler (ADCLKPS)** value, setting the **ADC clock prescaler (ADCLKPS)** parameter to 1, the default value, divides the result by 2.

ADC Module clock (ADCCLK = HSPCLK/ADCLKPS*2)/(CPS+1) in MHz

The clock to the ADC module and indicates the ADC operating clock speed.

Acquisition window prescaler (ACQ_PS)

This value does not directly alter the core clock speed of the ADC. It serves to determine the width of the sampling or acquisition period. The higher the value, the wider is the sampling period. The default value is 4.

Acquisition window size ((ACQ_PS+1)/ADCCLK) in micro seconds/channel

Acquisition window size determines for what time duration the sampling switch is closed. The width of SOC pulse is ADCTRL1[11:8] + 1 times the ADCLK period.

Offset

Enter the offset value.

Use external reference 2.048VExternal reference

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use a 2.048V external voltage reference.

Use external reference

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use an external voltage reference.

Continuous mode

When the ADC generates an end of conversion (EOC) signal, generate an ADCINT# interrupt whether the previous interrupt flag has been acknowledged or not.

ADC offset correction (OFFSET_TRIM: -256 to 255)

The 280x ADC supports offset correction via a 9-bit value that it adds or subtracts before the results are available in the ADC result registers. Timing for results is not affected. The default value is 0.

VREFHI VREFLO

When you disable the **Use external reference 2.048V** or **External reference** option, the ADC logic uses a fixed 0-volt to 3.3-volt input range and the software disables **VREFHI** and **VREFLO**. To interpret the ADC input as a ratiometric signal, select the **External reference** option. Then set values for the high voltage reference (**VREFHI**) and the low voltage reference (**VREFLO**). **VREFHI** uses the external ADCINA0 pin, and **VREFLO** uses the internal GND.

INT pulse control

Use this option to configure when the ADC sets ADCINTFLG .ADCINTx relative to the SOC and EOC Pulses. Select **Late interrupt pulse** or **Early interrupt pulse**.

SOC high priority

Use this option to enable and configure **SOC high priority mode** . In all in round robin mode, the default selection, the ADC services each SOC interrupt in a numerical sequence.

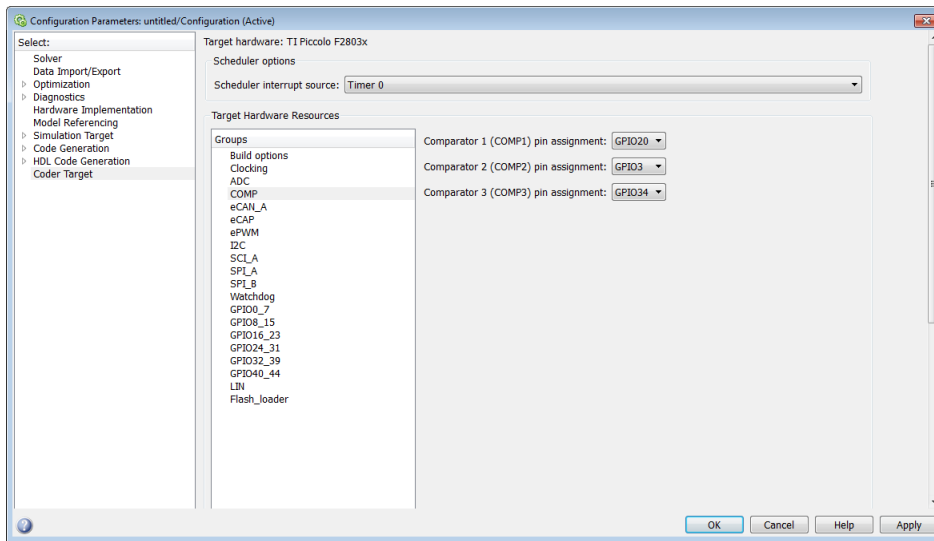
Choose one of the **high priority** selections to assign high priority to one or more of the SOC's. In this mode, the ADC operates in round robin mode until it receives a high priority SOC interrupt. The ADC finishes servicing the current SOC, services the high priority SOC's, and then returns to the next SOC in the round robin sequence.

For example, the ADC is servicing SOC8 when it receives a high priority interrupt on SOC1. The ADC completes servicing SOC8, services SOC1, and then services SOC9.

XINT2SOC external pin

Select the pin to which the ADC sends the XINT2SOC pulse.

COMP



Assigns COMP pins to GPIO pins.

Comparator 1 (COMP1) pin assignment

Select an option from the list — None,GPIO1, GPIO20, GPIO42.

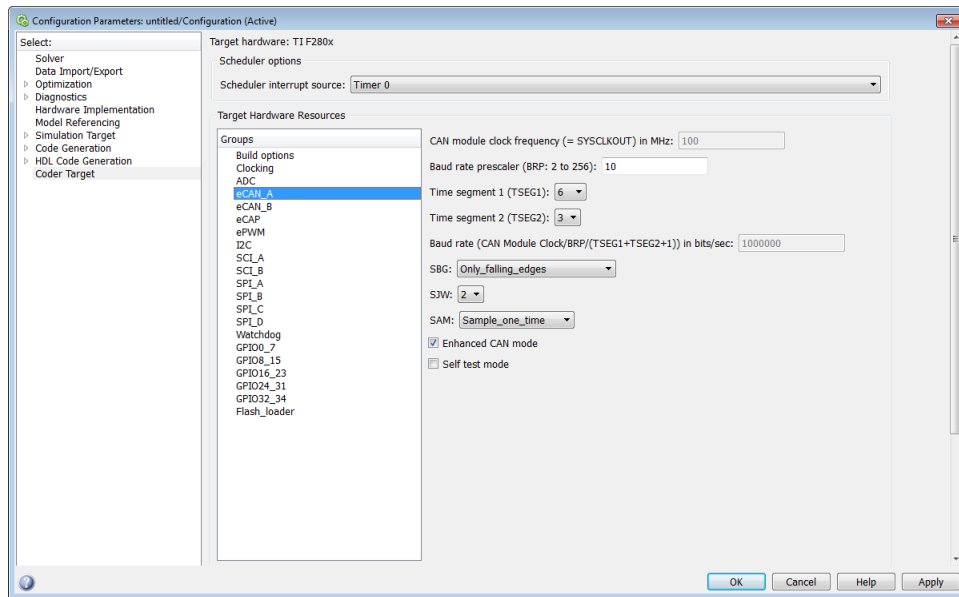
Comparator 2 (COMP2) pin assignment

Select an option from the list — None,GPIO3, GPIO21, GPIO34, GPIO43.

Comparator 3 (COMP3) pin assignment

Select an option from the list — None,GPIO34.

eCAN_A, eCAN_B



For more help on setting the timing parameters for the eCAN modules, refer to “Configuring Timing Parameters for CAN Blocks”. You can set the following parameters for the eCAN module:

CAN module clock frequency (= SYSCLKOUT) in MHz:

The clock to the enhanced CAN module. The CAN module clock frequency is equal SYSCLKOUT for processors such as c280x, c281x, c28044.

CAN module clock frequency (=SYSCLKOUT/2) in MHz

The clock to the enhanced CAN module. The CAN module clock frequency is equal to SYSCLKOUT/2 for processors such as piccolo, c2834x, c28x3x.

Baud rate prescaler (BRP: 2 to 256):

Value by which to scale the bit rate. Valid values are from 2 to 256.

Time segment 1 (TSEG1):

Sets the value of time segment 1, which, with **TSEG2** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG1** are from 1 through 16.

Time segment 2 (TSEG2):

Sets the value of time segment 2, which, with **TSEG1** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG2** are from 1 through 8.

Baud rate (CAN Module Clock/BRP/(TSEG1 + TSEG2 +1)) in bits/sec:

CAN module communication speed represented in bits/sec.

SBG

Sets the message resynchronization triggering. Options are `Only_falling_edges` and `Both_falling_and_rising_edges`.

SJW

Sets the synchronization jump width, which determines how many units of TQ a bit can be shortened or lengthened when resynchronizing.

SAM

Number of samples used by the CAN module to determine the CAN bus level. Selecting `Sample_one_time` samples once at the sampling point. Selecting `Sample_three_times` samples once at the sampling point and twice before at a distance of TQ/2. The CAN module makes a majority decision from the three points.

Enhanced CAN Mode

To enable time-stamping and to use **Mailbox Numbers** 16 through 31 in the C2000 eCAN blocks, enable this parameter. Texas Instruments documentation refers to this “HECC mode”.

Self test mode

If you set this parameter to `True`, the eCAN module goes to loopback mode. Loopback mode sends a “dummy” acknowledge message back without needing an acknowledge bit. The default is `False`.

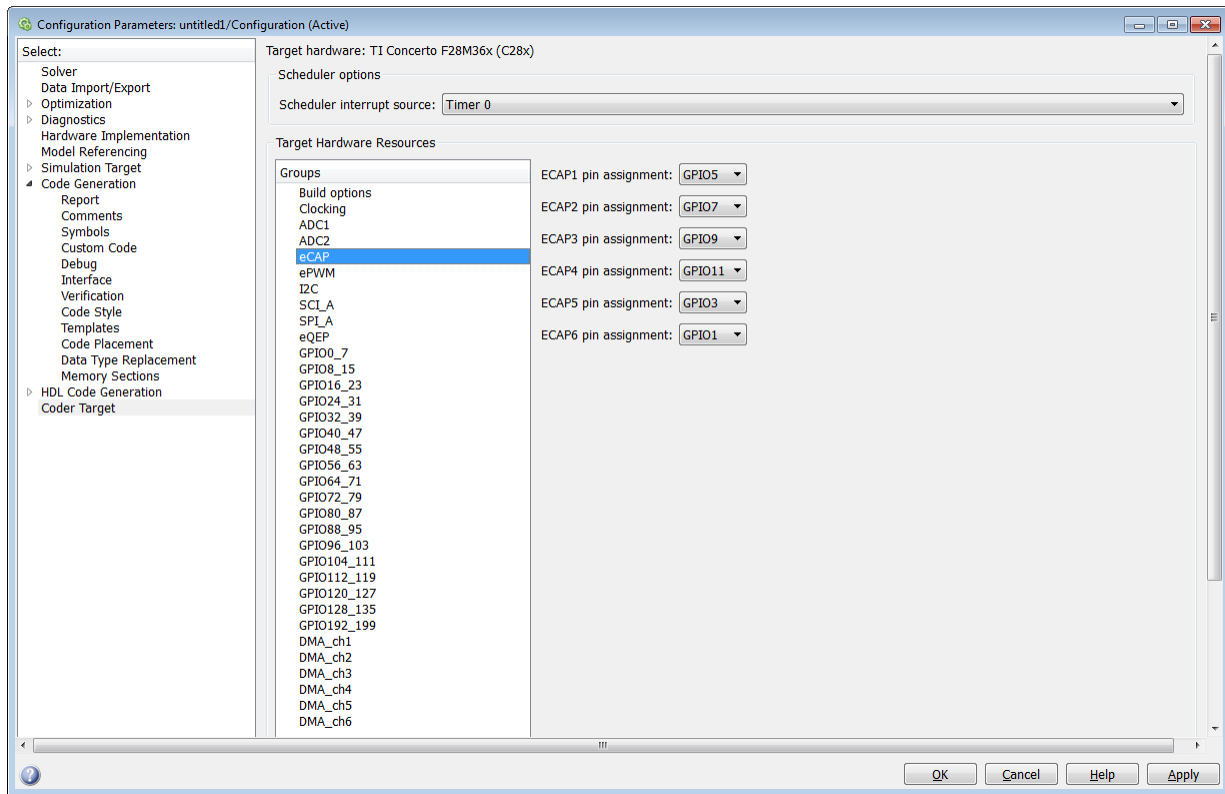
Pin assignment (Tx)

Assigns the CAN transmit pin to use with the eCAN_B module. Possible values are GPIO8, GPIO12, GPIO16, and GPIO20.

Pin assignment (Rx)

Assigns the CAN receive pin to use with the eCAN_B module. Possible values are GPIO10, GPIO13, GPIO17, and GPIO21.

eCAP



Assigns eCAP pins to GPIO pins.

ECAP1 pin assignment

Select an option from the list—None, GPIO5, or GPIO24 or GPIO34.

ECAP2 pin assignment

Select an option from the list—None, GPIO7, or GPIO25 or GPIO37.

ECAP3 pin assignment

Select an option from the list—None, GPIO9, or GPIO26.

ECAP4 pin assignment

Select an option from the list—None, GPIO11, or GPIO27.

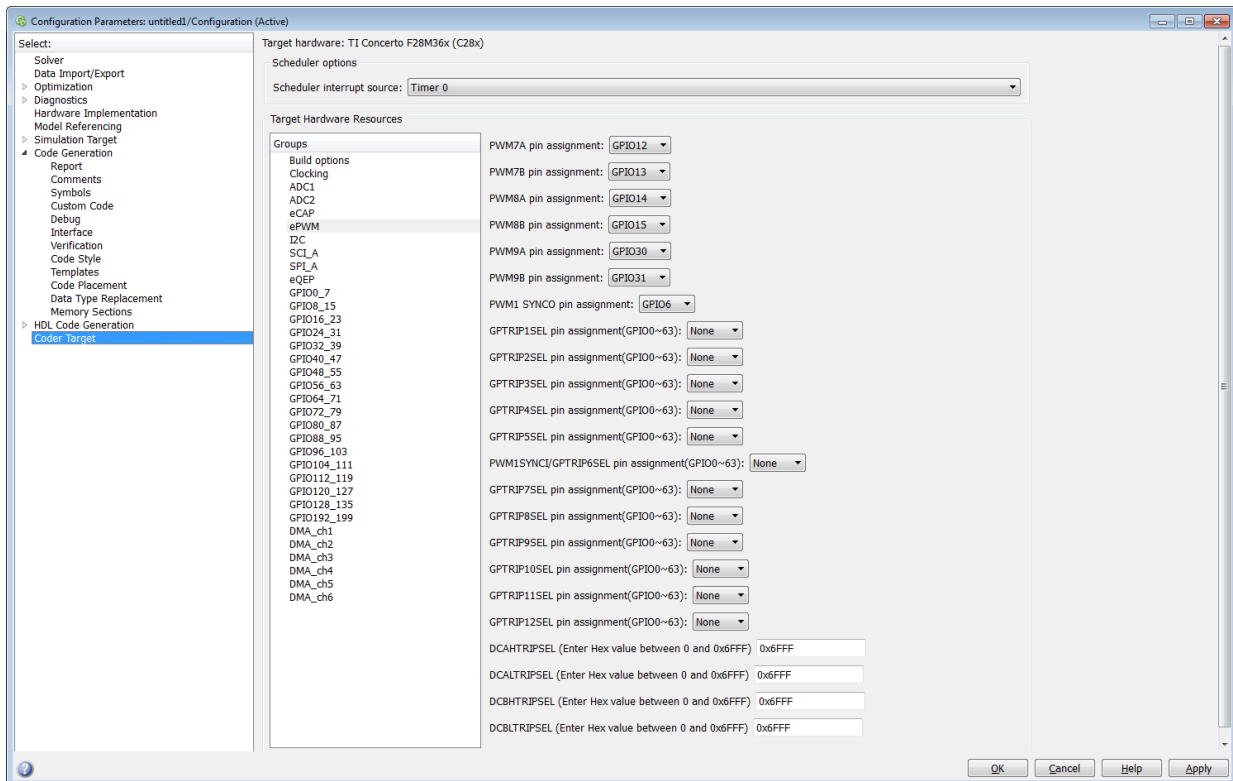
ECAP5 pin assignment

Select an option from the list—None, GPIO3, or GPIO48.

ECAP6 pin assignment

Select an option from the list—None, GPIO1, or GPIO49.

ePWM



Assigns ePWM signals to GPIO pins.

TZ1 pin assignment

Assigns the trip-zone input 1 (TZ1) to a GPIO pin. Choices are None (the default), GPIO12, and GPIO15.

TZ2 pin assignment

Assigns the trip-zone input 2 (TZ2) to a GPIO pin. Choices are None (the default), GPIO16, and GPIO28.

TZ3 pin assignment

Assigns the trip-zone input 3 (TZ3) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO29.

TZ4 pin assignment

Assigns the trip-zone input 4 (TZ4) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO28.

TZ5 pin assignment

Assigns the trip-zone input 5 (TZ5) to a GPIO pin. Choices are None (the default), GPIO16, and GPIO28.

TZ6 pin assignment

Assigns the trip-zone input 6 (TZ6) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO29.

Note The TZ# pin assignments are available only for TI F280x processors.

SYNCI pin assignment

Assigns the ePWM external sync pulse input (SYNCI) to a GPIO pin. Choices are None (the default), GPIO6, and GPIO32.

SYNCO pin assignment

Note SYNCI and SYNCO pin assignments are available for TI F28044, TI F280x, TI Delfino F2833x, TI Delfino F2834x, TI Piccolo F2802x, TI Piccolo F2803x, TI Piccolo F2806 processors.

Assigns the ePWM external sync pulse output (SYNCO) to a GPIO pin. Choices are None (the default), GPIO6, and GPIO33.

PWM#A, PWM#B, PWM#C pin assignment

The PWM # A, PWM # B, PWM # C pin assignment.

GPTRIP#SEL pin assignment

Assigns the ePWM trip-zone input to a GPIO pin.

Note The GPTRIP#SEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

PWM1SYNCl/ GPTRIP6SEL pin assignment

Assigns the ePWM sync pulse input (SYNCl) to a GPIO pin.

Note: The PWM1SYNCl/GPTRIP#SEL pin assignments are available only for TI Concerto F28M35x/F28M36x processors.

DCAHTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBHTRIPSEL (Enter Hex value between 0 and 0x6FFF)

Assigns the Digital Compare A High Trip Input to a GPIO pin.

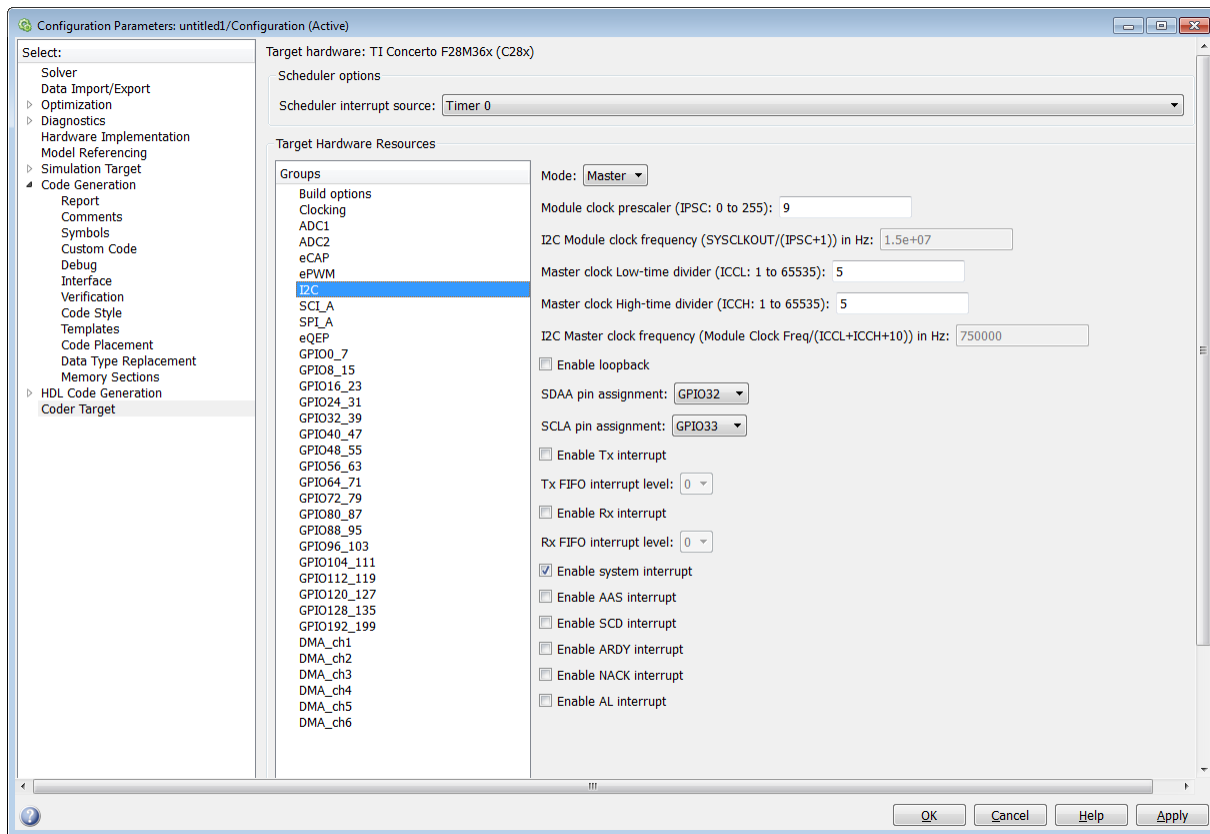
Note DCAHTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

DCALTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBLTRIPSEL (Enter Hex value between 0 and 0x6FFF)

Assigns the Digital Compare A High Trip Input to a GPIO pin.

Note The DCALTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

I2C



Report or set Inter-Integrated Circuit parameters. For more information, consult the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B available on the Texas Instruments Web site.

Mode

Configure the I2C module as **Master** or **Slave**.

If a module is an I2C master, it:

Initiates communication with slave nodes by sending the slave address and requesting data transfer to or from the slave.

Outputs the **Master clock frequency** on the serial clock line (SCL) line.

If a module is an I2C slave, it:

- Synchronizes itself with the serial clock line (SCL) line.
- Responds to communication requests from the master.

When **Mode** is **Slave**, you can configure the **Addressing format**, **Address register**, and **Bit count** parameters.

The **Mode** parameter corresponds to bit 10 (MST) of the I2C Mode Register (I2CMR).

Addressing format

If **Mode** is **Slave**, determine the addressing format of the I2C master, and set the I2C module to the same mode:

- **7-Bit Addressing**, the normal address mode.
- **10-Bit Addressing**, the expanded address mode.
- **Free Data Format**, a mode that does not use addresses. (If you **Enable loopback**, the Free data format is not supported.)

The **Addressing format** parameter corresponds to bit 3 (FDF) and bit 8 (XA) of the I2C Mode Register (I2CMR).

Own address register

If **Mode** is **Slave**, enter the 7-bit (0–127) or 10-bit (0–1023) address this I2C module uses while it is a slave.

This parameter corresponds to bits 9–0 (OAR) of the I2C Own Address Register (I2COAR).

Bit count

If **Mode** is **Slave**, set the number of bits in each data *byte* the I2C module transmits and receives. This value must match that of the I2C master.

This parameter corresponds to bits 2–0 (BC) of the I2C Mode Register (I2CMR).

Module clock prescaler (IPSC: 0 to 255):

If **Mode** is **Master**, configure the module clock frequency by entering a value 0–255.

Module clock frequency = I2C input clock frequency / (Module clock prescaler + 1)

The I2C specifications require a module clock frequency between 7 MHz and 12 MHz.

The *I2C input clock frequency* depends on the DSP input clock frequency and the value of the PLL Control Register divider (PLLCR). For more information on setting the PLLCR, consult the documentation for your specific Digital Signal Controller.

This **Module clock prescaler (IPSC: 0 to 255)**: corresponds to bits 7–0 (IPSC) of the I2C Prescaler Register (I2CPSC).

I2C Module clock frequency (SYSCLKOUT / (IPSC+1)) in Hz:

This field displays the frequency the I2C module uses internally. To set this value, change the **Module clock prescaler**.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721, on the Texas Instruments Web site.

I2C Master clock frequency (Module Clock Freq/(ICCL+ICCH+10)) in Hz:

This field displays the master clock frequency.

For more information about this value, consult the “Clock Generation” section in the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

Master clock Low-time divider (ICCL: 1 to 65535):

When **Mode** is **Master**, this divider determines the duration of the low state of the SCL line on the I2C-bus.

The low-time duration of the master clock = $T_{\text{mod}} \times (\text{ICCL} + d)$.

For more information, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCL) of the Clock Low-Time Divider Register (I2CCLKL).

Master clock High-time divider (ICCH: 1 to 65535):

When **Mode** is **Master**, this divider determines the duration of the high state on the serial clock pin (SCL) of the I2C-bus.

The high-time duration of the master clock = $T_{\text{mod}} \times (\text{ICCL} + d)$.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A, SPRUH22f, SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCH) of the Clock High-Time Divider Register (I2CCLKH).

Enable loopback

When **Mode** is **Master**, enable or disable digital loopback mode. In digital loopback mode, I2CDXR transmits data over an internal path to I2CDRR, which receives the data after a configurable delay.

The delay, measured in DSP cycles, equals (I2C input clock frequency/module clock frequency) x 8.

While **Enable loopback** is enabled, free data format addressing is not supported.

This parameter corresponds to bit 6 (DLB) of the I2C Mode Register (I2CMODR).

SDAA pin assignment

Select the GPIO pin to use on the SDAA boot function.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

SCLA pin assignment

Select the GPIO pin to use on the SCLA boot function.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

Enable Tx interrupt

This parameter corresponds to bit 5 (TXFFIENA) of the I2C Transmit FIFO Register (I2CFFTX).

Tx FIFO interrupt level

This parameter corresponds to bits 4–0 (TXFFIL4-0) of the I2C Transmit FIFO Register (I2CFFTX).

Enable Rx interrupt

This parameter corresponds to bit 5 (RXFFIENA) of the I2C Receive FIFO Register (I2CFFRX).

Rx FIFO interrupt level

This parameter corresponds to bit 4–0 (RXFFIL4-0) of the I2C Receive FIFO Register (I2CFFRX).

Enable system interrupt

Select this parameter to display and individually configure the following five Basic I2C Interrupt Request parameters in the Interrupt Enable Register (I2CIER):

- Enable AAS interrupt
- Enable SCD interrupt
- Enable ARDY interrupt
- Enable NACK interrupt
- Enable AL interrupt

Enable AAS interrupt

Enable the addressed-as-slave interrupt.

When enabled, the I2C module generates an interrupt (AAS bit = 1) upon receiving one of the following:

- Its **Own address register**
- A general call (all zeros)
- A data byte is in free data format

When enabled, the I2C module clears the interrupt (AAS = 0) upon receiving one of the following:

- Multiple START conditions (7-bit addressing mode only)
- A slave address that is different from **Own address register** (10-bit addressing mode only)
- A NACK or a STOP condition

This parameter corresponds to bit 6 (AAS) of the Interrupt Enable Register (I2CIER).

Enable SCD interrupt

Enable stop condition detected interrupt.

When enabled, the I2C module generates an interrupt (SCD bit = 1) when the CPU detects a stop condition on the I2C bus.

When enabled, the I2C module clears the interrupt (SCD = 0) upon one of the following events:

- The CPU reads the I2CISRC while it indicates a stop condition
- A reset of the I2C module
- Someone manually clears the interrupt

This parameter corresponds to bit 5 (SCD) of the Interrupt Enable Register (I2CIER).

Enable ARDY interrupt

Enable register-access-ready interrupt enable bit.

When enabled, the I2C module generates an interrupt (ARDY bit = 1) when the previous address, data, and command values in the I2C module registers have been used and new values can be written to the I2C module registers.

This parameter corresponds to bit 2 (ARDY) of the Interrupt Enable Register (I2CIER).

Enable NACK interrupt

Enable no acknowledgment interrupt enable bit.

When enabled, the I2C module generates an interrupt (NACK bit = 1) when the module is a transmitter in master or slave mode and it receives a NACK condition.

This parameter corresponds to bit 1 (NACK) of the Interrupt Enable Register (I2CIER).

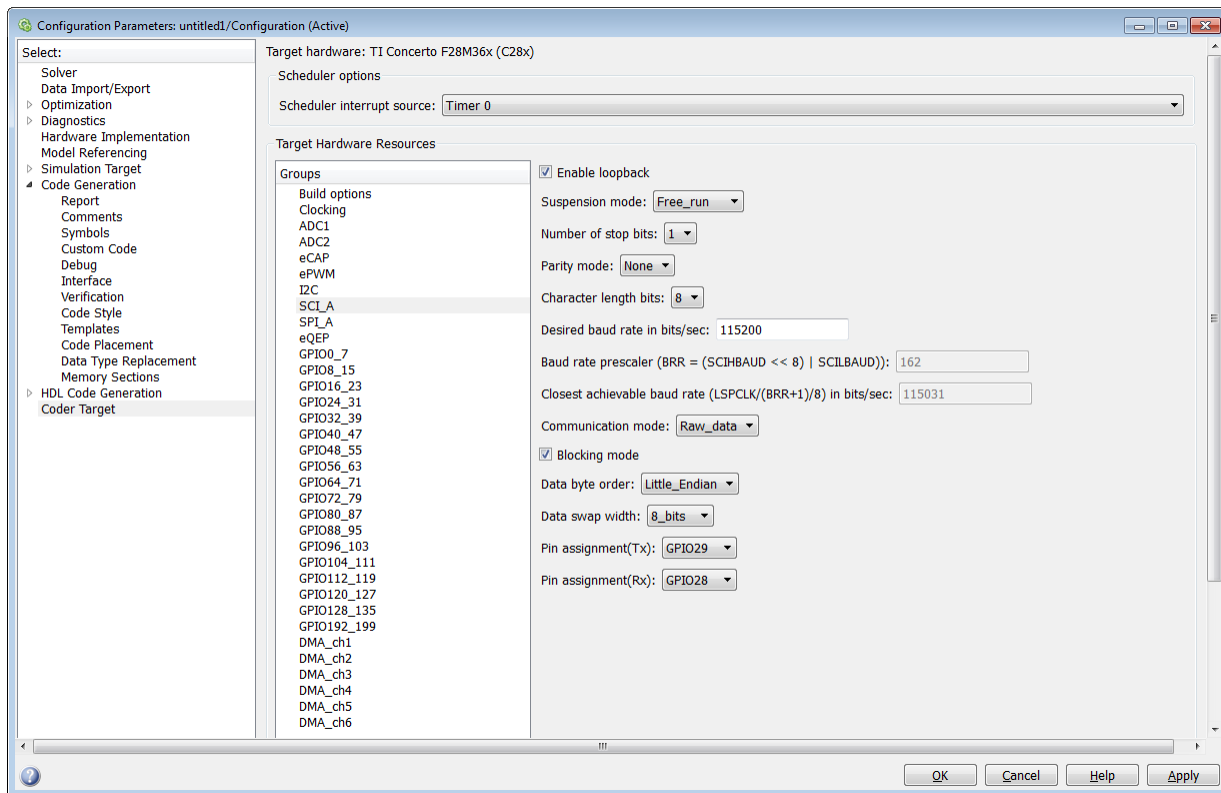
Enable AL interrupt

Enable arbitration-lost interrupt.

When enabled, the I2C module generates an interrupt (AL bit = 1) when the I2C module is operating as a master transmitter and loses an arbitration contest with another master transmitter.

This parameter corresponds to bit 0 (AL) of the Interrupt Enable Register (I2CIER).

SCI_A, SCI_B, SCI_C



The serial communications interface parameters you can set for module A. These parameters are:

Enable loopback

Select this parameter to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, a C28x DSP Tx pin is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

Baud rate

Baud rate for transmitting and receiving data. Select from 115200 (the default), 57600, 38400, 19200, 9600, 4800, 2400, 1200, 300, and 110.

Suspension mode

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the suspension mode determines whether to perform the program instruction. Available options are **Hard_abort**, **Soft_abort**, and **Free_run**. **Hard_abort** stops the program immediately. **Soft_abort** stops when the current receive/transmit sequence is complete. **Free_run** continues running regardless of the breakpoint.

Number of stop bits

Select whether to use 1 or 2 stop bits.

Parity mode

Type of parity to use. Available selections are **None**, **Odd parity**, or **Even parity**. **None** disables parity. **Odd** sets the parity bit to one if you have an odd number of ones in your bytes, such as 00110010. **Even** sets the parity bit to one if you have an even number of ones in your bytes, such as 00110011.

Character length bits

Length in bits of each transmitted or received character, set to 8 bits.

Desired baud rate in bits/sec

The desired baud rate specified by the user.

Baud rate prescaler (BRR = (SCIHBAUD << 8) | SCILBAUD)

The baud rate prescaler.

Closest achievable baud rate (LSPCLK/(BRR+1)/8) in bits/sec

The closest achievable baud rate calculated based on LSPCLK and BRR.

Communication mode

Select **Raw_data** or **Protocol** mode. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlock conditions do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

When you select protocol mode, some handshaking between host and processor occurs. The transmitting side sends \$SND to indicate it is ready to transmit. The receiving side sends back \$RDY to indicate it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include:

- Avoids deadlock

- Determines whether data is received without errors (checksum)
- Determines whether data is received by processor
- Determines time consistency; each side waits for its turn to send or receive

Note Deadlocks can occur if one SCI Transmit block tries to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

Blocking mode

If this option is set to `True`, system waits until data is available to read (when data length is reached). If this option is set to `False`, system checks FIFO periodically (in polling mode) for data to read. If data is present, the block reads and outputs the contents. When data is not present, the block outputs the last value and continues.

Data byte order

Select `Little Endian` or `Big Endian`, to match the endianness of the data being moved.

Data swap width

Select `8_bits` or `16_bits`, to match the width of the data being moved by the data swap operation. When you set **Data byte order** to `Big Endian`, the only available option for **Data swap width** is `8_bits`.

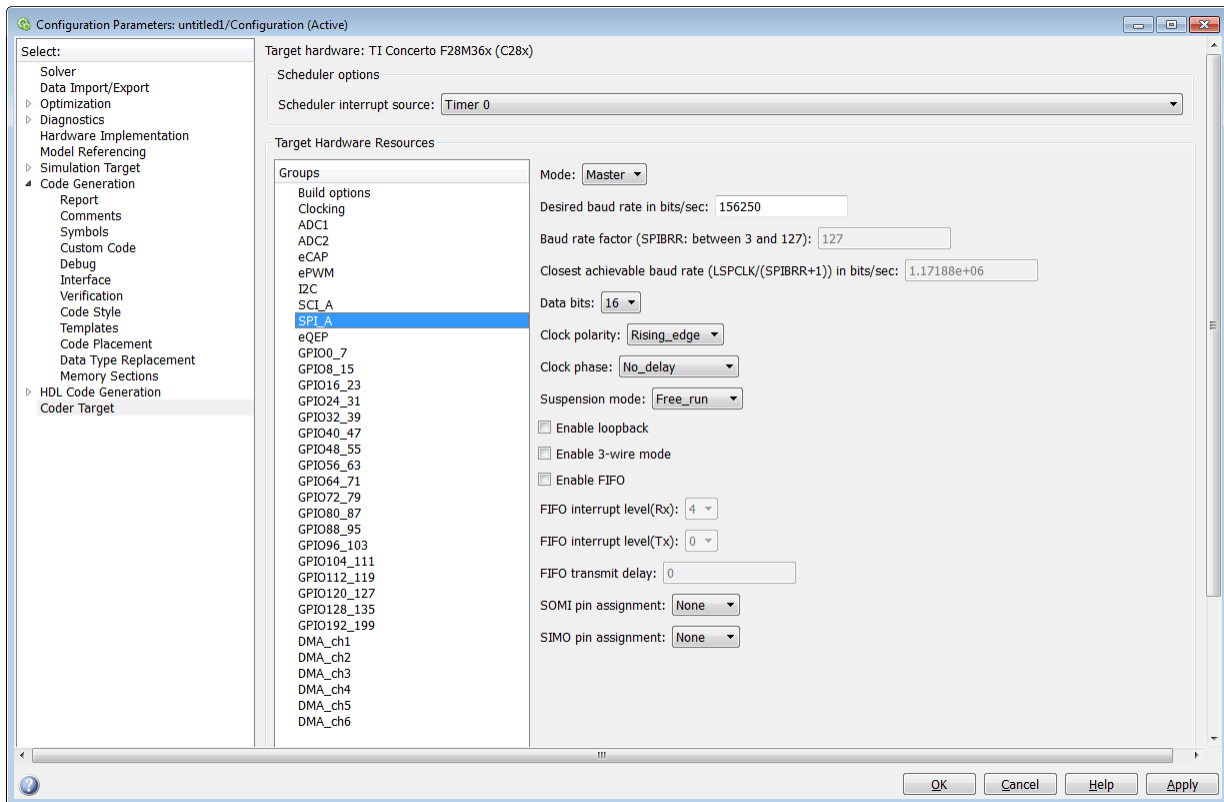
Pin assignment (Tx)

Assigns the SCI transmit pin to use with the SCI module.

Pin assignment (Rx)

Assigns the SCI receive pin to use with the SCI module.

SPI_A, SPI_B, SPI_C, SPI_D



The serial peripheral interface parameters you can set for the A module. These parameters are:

Mode

Set to **Master** or **Slave**.

Desired baud rate in bits/sec

The desired baud rate specified by the user.

Baud rate factor (SPIBRR: between 3 and 127)

To set the **Baud rate factor**, search for “Baud Rate Determination” and “SPI Baud Rate Register (SPIBRR) Bit Descriptions” in *TMS320x28xx, 28xxx DSP Serial Peripheral Interface (SPI) Reference Guide*, Literature Number: SPRU059, available on the Texas Instruments Web Site.

Closest achievable baud rate (LSPCLK/(SPIBRR+1)) in bits/sec

The closest achievable baud rate calculated based on LSPCLK and SPIBRR.

Data bits

Length in bits from 1 to 16 of each transmitted or received character. For example, if you select 8, the maximum data that can be transmitted using SPI is 2^{8-1} . If you send data greater than this value, the buffer overflows.

Clock polarity

Select `Rising_edge` or `Falling_edge`.

Clock phase

Select `No_delay` or `Delay_half_cycle`.

Suspension mode

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the selected suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive or transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

Enable loopback

Select this option to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, the Tx pin on a C28x DSP is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

Enable 3-wire mode

Enable SPI communication over three pins instead of the normal four pins.

Enable FIFO

Set true or false.

FIFO interrupt level (Rx)

Set level for receive FIFO interrupt. Select 0 through 16.

FIFO interrupt level (Tx)

Set level for transmit FIFO interrupt. Select 0 through 16.

FIFO transmit delay

Enter FIFO transmit delay (in processor clock cycles) to pause between data transmissions. Enter an integer.

CLK pin assignment

Assigns the SPI something (CLK) to a GPIO pin. Choices are **None** (default), **GPI014**, or **GPI026**.

CLK pin assignment is not available for TI Concerto F28M35x/F28M36x processors.

SOMI pin assignment

Assigns the SPI something (SOMI) to a GPIO pin. Choices are **None** (default), **GPI013**, or **GPI025**.

STE pin assignment

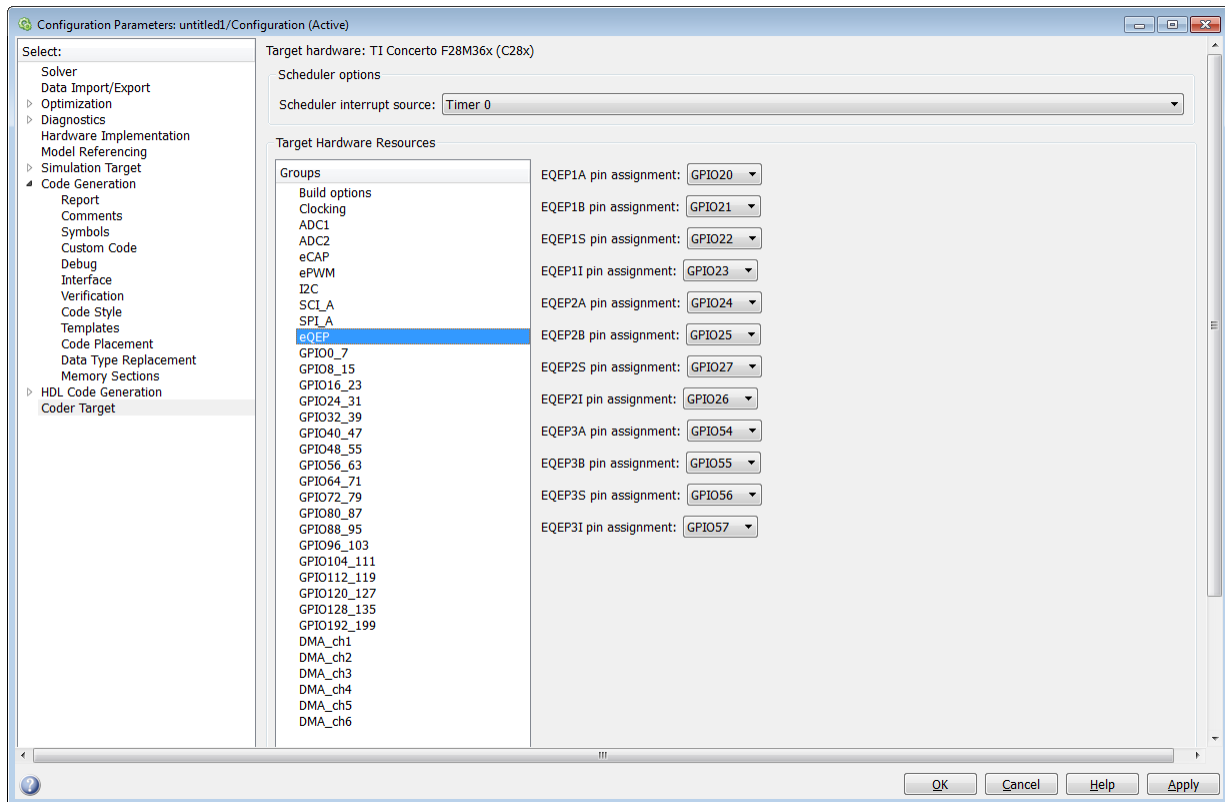
Assigns the SPI something (STE) to a GPIO pin. Choices are **None** (default), **GPI015**, or **GPI027**.

STE pin assignment is not available for TI Concerto F28M35x/ F28M36x processors.

SIMO pin assignment

Assigns the SPI something (SIMO) to a GPIO pin. Choices are **None** (default), **GPI012**, or **GPI024**.

eQEP



Assigns eQEP pins to GPIO pins.

EQEP1A pin assignment

Select an option from the list—None, GPIO20, GPIO50, GPIO64, GPIO106.

EQEP1B pin assignment

Select an option from the list—None, GPIO21, GPIO51, GPIO65, GPIO107.

EQEP1S pin assignment

Select an option from the list—None, GPIO22, GPIO52, GPIO66, GPIO108.

EQEP1I pin assignment

Select an option from the list—None, GPIO23, GPIO53, GPIO67, GPIO109.

EQEP2A pin assignment

Select an option from the list—None, GPIO24, GPIO54, GPIO66, GPIO110. The pin numbers shown in the list vary based on the processor selected.

EQEP2B pin assignment

Select an option from the list—None, GPIO25, GPIO55, GPIO67, GPIO111. The pin numbers shown in the list vary based on the processors selected.

EQEP2S pin assignment

Select an option from the list—None, GPIO27, GPIO31, GPIO57, GPIO67, GPIO113.

EQEP2I pin assignment

Select an option from the list—None, GPIO26, GPIO30, GPIO56, GPIO64, GPIO112.

EQEP3A pin assignment

Select an option from the list—None or GPIO54, or GPIO112. This parameter is available only for TI Concerto F28M36x (C28x) processors.

EQEP3B pin assignment

Select an option from the list—None or GPIO55, or GPIO113. This parameter is available only for TI Concerto F28M36x (C28x) processors.

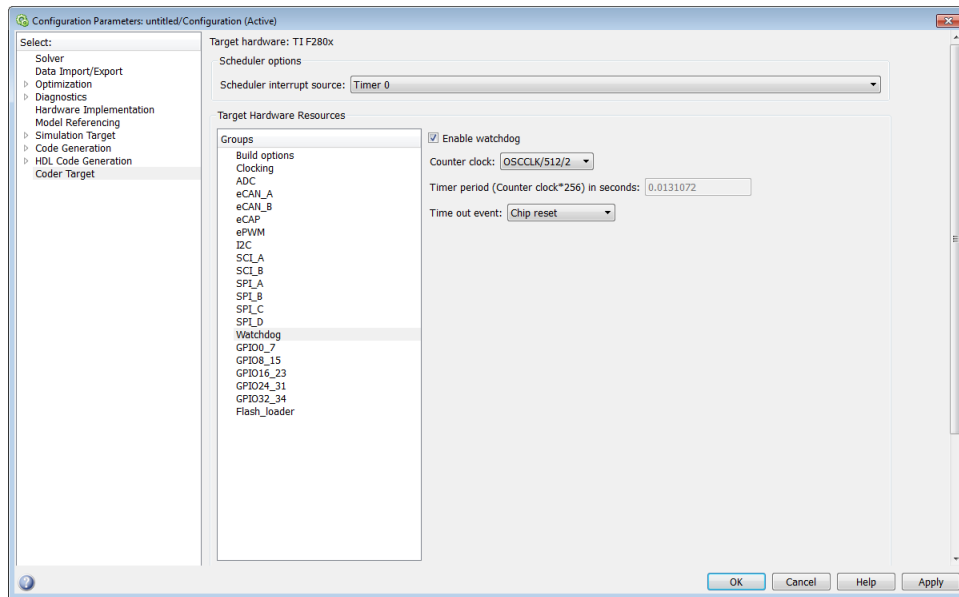
EQEP3S pin assignment

Select an option from the list—None, or GPIO56, or GPIO110

EQEP3I pin assignment

Select an option from the list—None, or GPIO57, or GPIO111

Watchdog



When enabled, if the software fails to reset the watchdog counter within a specified interval, the watchdog resets the processor or generates an interrupt. This feature enables the processor to recover from some fault conditions.

For more information, locate the *Data Manual* or *System Control and Interrupts Reference Guide* for your processor on the Texas Instruments Web site.

Enable watchdog

Enable the watchdog timer module.

This parameter corresponds to bit 6 (WDDIS) of the Watchdog Control Register (WDCR) and bit 0 (WDOVERRIDE) of the System Control and Status Register (SCSR).

Counter clock

Set the watchdog timer period relative to OSCCLK/512.

This parameter corresponds to bits 2–0 (WDPS) of the Watchdog Control Register (WDCR).

Timer period (Counter clock*256) in seconds

This field displays the timer period in seconds. This value automatically updates when you change the **Counter clock** parameter.

Time out event

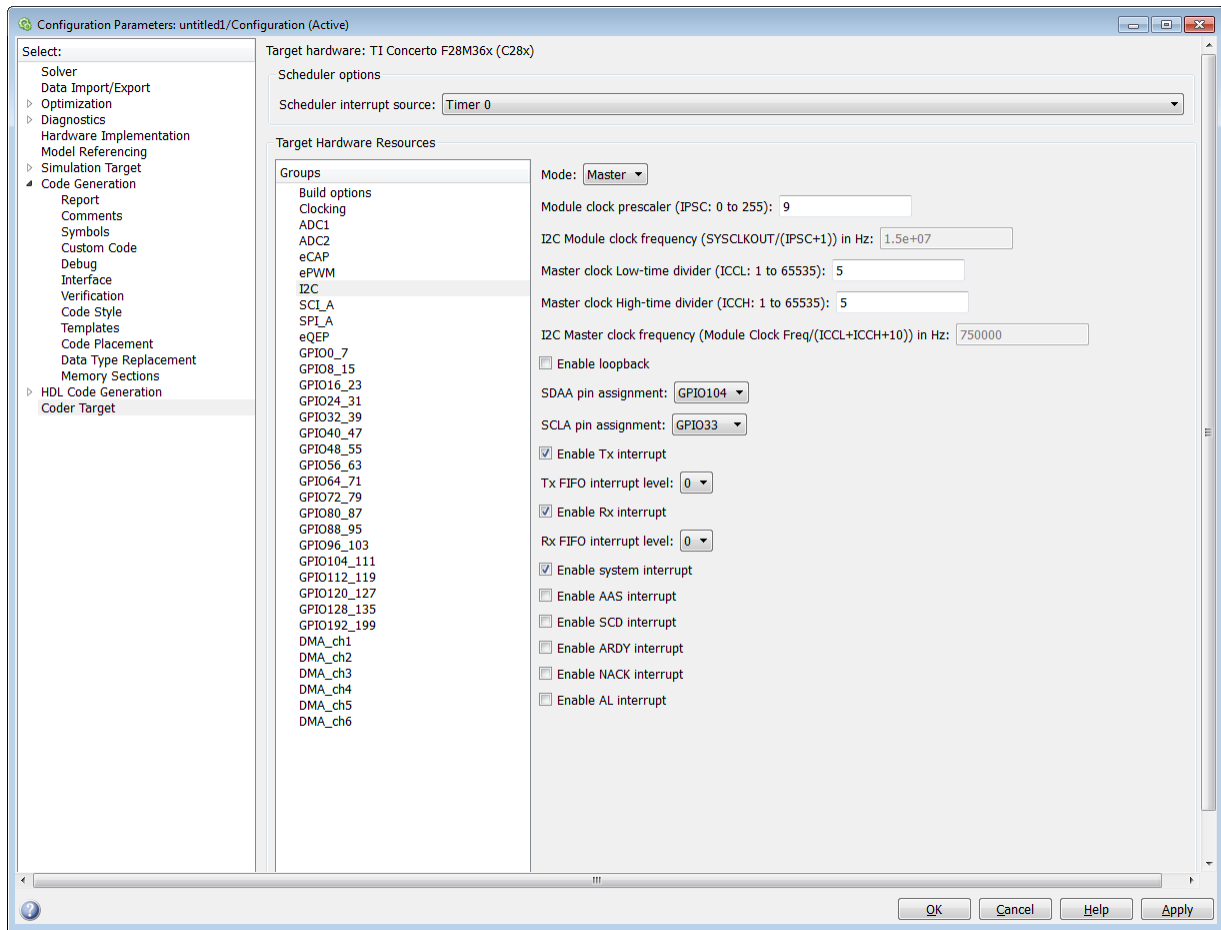
Configure the watchdog to reset the processor or generate an interrupt when the software fails to reset the watchdog counter:

- Select **Chip reset** to generate a signal that resets the processor (WDRST signal) and disable the watchdog interrupt signal (WDINT signal).
- Select **Raise WD Interrupt** to generate a watchdog interrupt signal (WDINT signal) and disable the reset processor signal (WDRST signal). This signal can be used to wake the device from an IDLE or STANDBY low-power mode.

This parameter corresponds to bit 1 (WDENINT) of the System Control and Status Register (SCSR).

GPIO

3 Configuration Parameters



GPIO Use the GPIO pins for digital input or output by connecting to one of the three peripheral I/O ports.

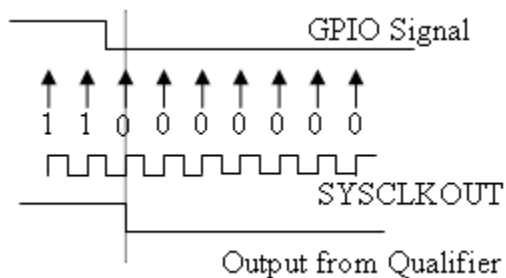
The range of GPIO pins for different processors is given below:

Processors	GPIO Pin Values
C281x	GPIOA, GPIOB, GPIOD, GPIOE, GPIOF, and GPIOG
F2803x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44

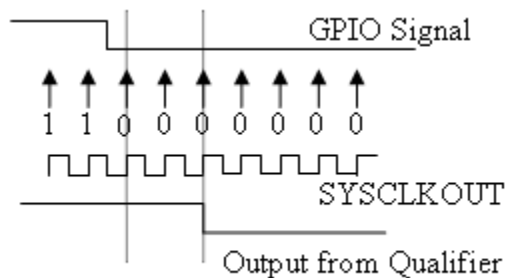
Processors	GPIO Pin Values
F2806x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44, GPIO50–55, GPIO56–58
F2823x, F2833x, and C2834x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48–55, GPIO56–63
C2801x, F2802x, F28044, F280x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–34
F28M35x (C28x)	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48_55, GPIO56–63, GPIO68–71, GPIO128–135
F28M36x (C28x)	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO40–47, GPIO48–55, GPIO56–63, GPIO64–71, GPIO72–79, GPIO80–87, GPIO88–95, GPIO96–103, GPIO104–111, GPIO112–119, GPIO120–127, GPIO128–135, GPIO192–199.

Each pin selected for input offers four signal qualification types:

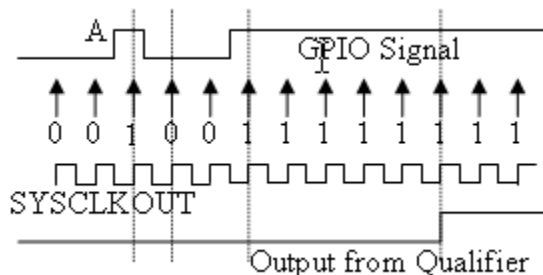
- **Sync to SYSCLKOUT only** — This setting is the default for all pins at reset. Using this qualification type, the input signal is synchronized to the system clock SYSCLKOUT. The following figure shows the input signal measured on each tick of the system clock, and the resulting output from the qualifier.



- **Qualification using 3 samples** — This setting requires three consecutive cycles of the same value for the output value to change. The following figure shows that, in the third cycle, the GPIO value changes to 0, but the qualifier output is still 1 because it waits for three consecutive cycles of the same GPIO value. The next three cycles all have a value of 0, and the output from the qualifier changes to 0 immediately after the third consecutive value is received.



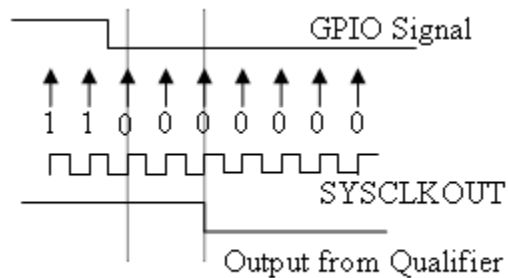
- Qualification using 6 samples** — This setting requires six consecutive cycles of the same GPIO input value for the output from the qualifier to change. In the following figure, the glitch **A** does not alter the output signal. When the glitch occurs, the counting begins, but the next measurement is low again, so the count is ignored. The output signal does not change until six consecutive samples of the high signal are measured.



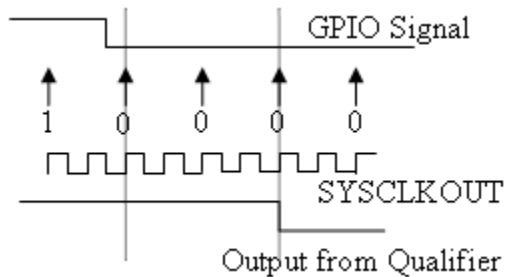
Qualification sampling period prescaler

Visible only when a setting for **Qualification type for GPIO [pin#]** is selected. The qualification sampling period prescaler, with possible values of 0 to 255, calculates the frequency of the qualification samples or the number of system clock ticks per sample. The formula for calculating the qualification sampling frequency is $\text{SYSCLKOUT}/(2 * \text{Prescaler})$, except for zero. When **Qualification sampling period prescaler=0**, a sample is taken every SYSCLKOUT clock tick. For example, a prescale setting of 0 means that a sample is taken on each SYSCLKOUT tick.

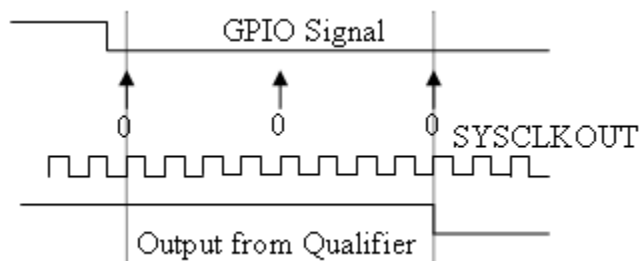
The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to **Qualification** using 3 samples. In this case, the **Qualification sampling period prescaler=0**:



In the next figure **Qualification sampling period prescaler=1**. A sample is taken every two clock ticks, and the **Qualification type** is set to **Qualification** using 3 samples. The output signal changes much later than if **Qualification sampling period prescaler=0**.



In the following figure, **Qualification sampling period prescaler=2**. Thus, a sample is taken every four clock ticks, and the **Qualification type** is set to **Qualification** using 3 samples.



- Asynchronous

Using this qualification type, the signal is synchronized to an asynchronous event initiated by software (CPU) via control register bits.

Qualification sampling period

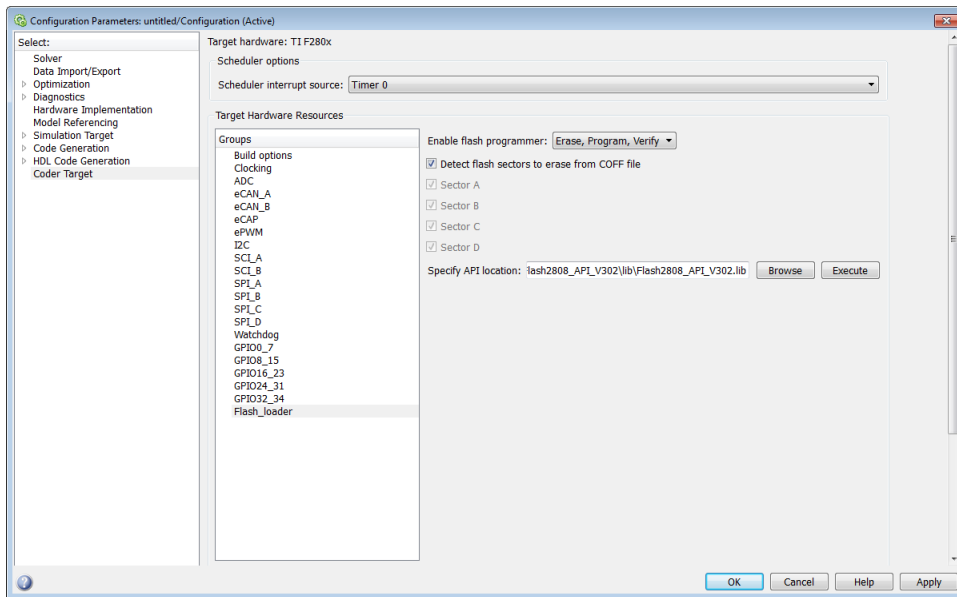
Enter the qualification sampling period.

GPIOA, GPIOB, GPIOD, GPIOE input qualification sampling period

GPIO# Pull Up Disabled

Select this check box to disable the GPIO pull up register. This option is available only for TI Concerto F28M35x/F28M36x processors.

Flash_loader



You can use `Flash_loader` to:

- Automatically erase program generated code to flash memory on the target when you build the code.
- Manually erase, program, or verify specific flash memory sectors.

To use this feature, download and install the TI Flash API plugin from the TI Web site.

For more information, consult the “Programming Flash Memory” topic or the `*_API_Readme.pdf` file included in the *TI Flash API* downloadable zip file.

Enable Flash Programmer

Enable the flash programmer by selecting a task for it to perform when you click **Execute** or build the software.

To program the flash memory when you build the software, select **Erase**, **Program**, **Verify**.

Detect Flash sectors to erase from COFF file

When enabled, the flash programmer erases all of the flash sectors defined by the COFF file.

Sector A, Sector B, Sector C...

When **Detect Flash sectors to erase from COFF file** is disabled, you can select the specific sector to erase.

Specify API location

Specify the folder path of the TI flash API executable you downloaded and installed on your computer.

Use **Browse** to locate the file or enter the path in the text box.

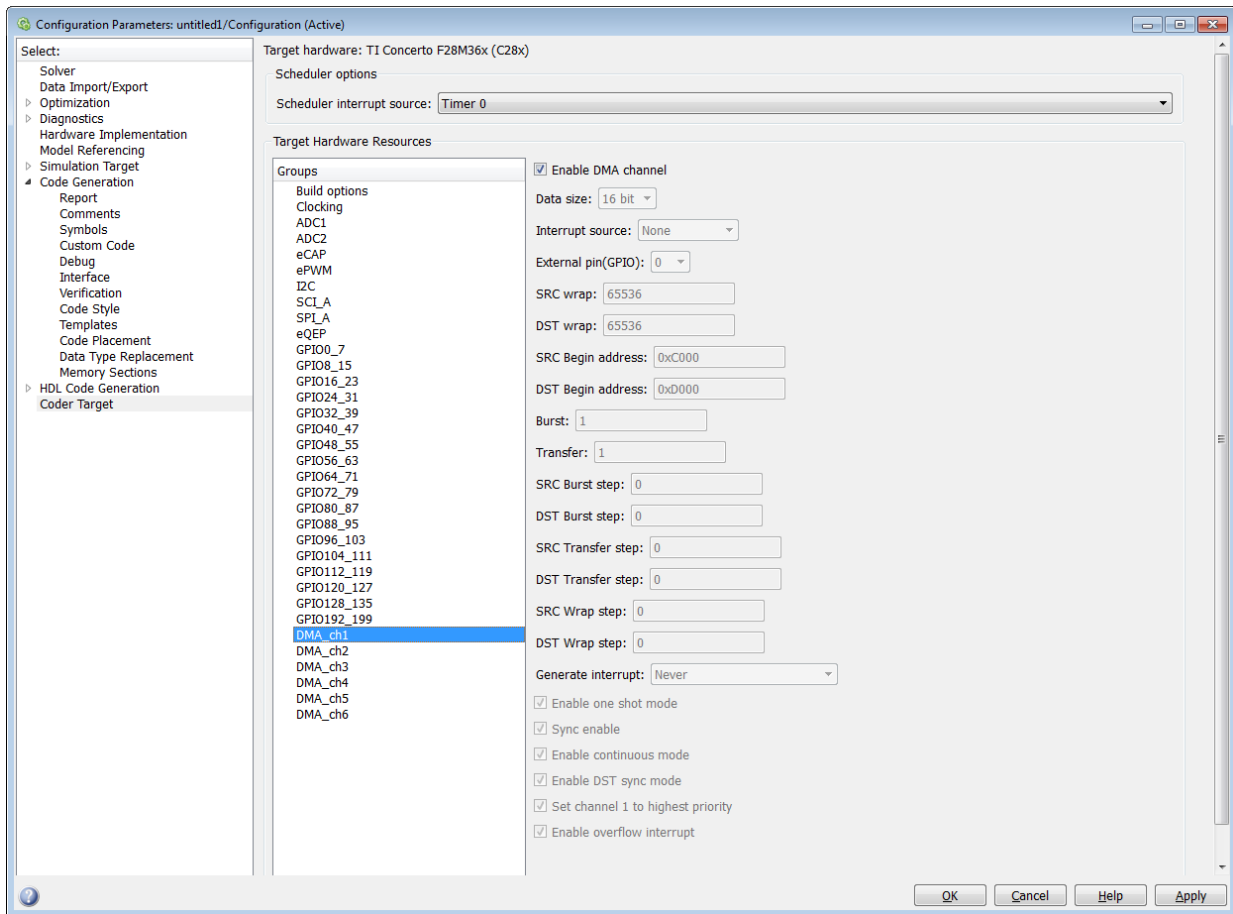
For example,

```
C:\TI\controlSUITE\libs\utilities\flash_api\2806x\v100\lib\2806x_BootROM_API_TABLE_Symbols_fpu32.lib
```

Execute

Click this button to initiate the task selected in **Enable Flash Programmer**.

DMA_ch[#]



The Direct Memory Access module transfers data directly between peripherals and memory using a dedicated bus, increasing overall system performance.

You can individually enable and configure each DMA channel.

The DMA module services are event driven. Using the Interrupt source and **External pin (GPIO)** parameters, you can configure a wide range of peripheral interrupt event triggers.

To avoid error messages, open the Coder Target > Target Hardware Resources, select the **Peripherals** tab, and *disable* the same DMA channel number.

For more information, consult the *TMS320x2833x, 2823x/ TMS320F28M3x Direct Memory Access (DMA) Module Reference Guide*, Literature Number: SPRUFB8A/SPRUH22F/SPRUHE8B.

Also consult the *Increasing Data Throughput using the TMS320F2833x DSC DMA* training presentation (requires login), both available from the TI Web site.

Enable DMA channel

Enable this parameter to edit the configuration of a specific DMA channel.

This parameter does not have a corresponding bit or register.

Data size

Select the size of the data bit transfer: **16 bit** or **32 bit**.

The DMA read/write data buses are 32 bits wide. 32-bit transfers have twice the data throughput of a 16-bit transfer.

When providing DMA service to McBSP, set **Data size** to **16 bit**.

The following parameters are based on a 16-bit word size. If you set **Data size** to **32 bit**, double the value of the following parameters:

- Size: Burst
- Source: Burst step
- Source: Transfer step
- Source: Wrap step
- Destination: Burst step
- Destination: Transfer step
- Destination: Wrap step

Data size corresponds to bit 14 (DATASIZE) in the Mode Register (MODE).

Interrupt source

Select the peripheral interrupt that triggers a DMA burst for the specified channel.

Select SEQ1INT or SEQ2INT to configure the ADC interrupt as interrupt source.

Select XINT1, XINT2, or XINT13 to configure GPIO pin 0 to 31 as an external interrupt source. Select XINT3 to XINT7 to configure GPIO pin 32 to 63 as an external interrupt source.

For more information about configuring XINT, consult the following references:

- *TMS320x2833x, 2823x External Interface (XINTF) User's Guide*, Literature Number: SPRU949, available on the TI Web site.
- *TMS320x2833x System Control and Interrupts*, Literature Number: SPRUFB0, available on the TI Web site.
- *TMS320F28M3x*, Literature Number: SPRUH22F/ SPRUHE8B available on the TI Web site.
- The and block reference sections.

Drop-down menu items from TINT0 to MREVTB may require manual configuration.

Select ePWM1SOCA through ePWM6SOCB to configure the ePWM interrupt as an interrupt source. Note that not all revisions of the TMS320F2833x silicon provide ePWM interrupts as sources for DMA transfers. For more information about silicon revisions consult the following reference:

TMS320x2833x, 2823x Silicon Errata/ TMS320F28M3x, Literature Number: SPRZ272/ SPRUH22F/ SPRUHE8B, available on the TI Web site.

The **Interrupt source** parameter corresponds to bit 4-0 (PERINTSEL) in the Mode Register (MODE).

External pin(GPIO)

When you set **Interrupt source** is set to an external interface (XINT[#]), specify the GPIO pin number from which the interrupt originates.

This parameter corresponds to the GPIO XINTn, XNMI Interrupt Select (GPIOXINTnSEL, GPIOXNMISEL) Registers.

For more information, consult the *TMS320x2833x / TMS320F28M3x System Control and Interrupts Reference Guide*, Literature Number SPRUFB0/ SPRUH22F/ SPRUHE8B available from the TI Web site.

SRC wrap

Specify the number of bursts before returning the current source address pointer to the **Source Begin Address** value. To disable wrapping, enter a value for **SRC wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (SRC_WRAP_SIZE) in the Source Wrap Size Register (SRC_WRAP_SIZE).

DST wrap

Specify the number of bursts before returning the current destination address pointer to the **Destination Begin Address** value. To disable wrapping, enter a value for **DST wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (DST_WRAP_SIZE) in the Destination Wrap Size Register (DST_WRAP_SIZE).

SRC Begin address

Set the starting address for the current source address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **SRC wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Source Begin Register (SRC_BEG_ADDR).

DST Begin address

Set the starting address for the current destination address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **DST wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Destination Begin Register (DST_BEG_ADDR).

Burst

Specify the number of 16-bit words in a burst, from 1 to 32. The DMA module must complete a burst before it can service the next channel.

Set the **Burst** value for the peripheral the DMA module is servicing. For the ADC, the value equals the number of ADC registers used, up to 16. For multichannel buffered serial ports (McBSP), which lack FIFOs, the value is 1.

For RAM, the value can range from 1 to 32.

This parameter corresponds to bits 4-0 (BURSTSIZE) in the Burst Size Register (BURST_SIZE).

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

Transfer

Specify the number of bursts in a transfer, from 1 to 65536.

This parameter corresponds to bits 15-0 (TRANSFERSIZE) in the Transfer Size Register (TRANSFER_SIZE).

SRC Burst step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually.

Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (SRCBURSTSTEP) in the Source Burst Step Size Register (SRC_BURST_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

DST Burst step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually. Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (DSTBURSTSTEP) in the Destination Burst Step Size Register (DST_BURST_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

SRC Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (SRCTRANSFERSTEP) Source Transfer Step Size Register (SRC_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **SRC wrap** is enabled) the corresponding source **Transfer step** does not alter the results.

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

DST Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (DSTTRANSFERSTEP) Destination Transfer Step Size Register (DST_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **DST wrap** is enabled) the corresponding destination **Transfer step** does not alter the results.

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

SRC Wrap step

Set the number of 16-bit words by which to increment or decrement the SRC_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Source Wrap Step Size Registers (SRC_WRAP_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to **32 bit**, double the value of this parameter.

DST Wrap step

Set the number of 16-bit words by which to increment or decrement the DST_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Destination Wrap Step Size Registers (DST_WRAP_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to **32 bit**, double the value of this parameter.

Generate interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via the PIE at the beginning or end of a data transfer.

This parameter corresponds to bit 15 (CHINTE) and bit 9 (CHINTMODE) in the Mode Register (MODE).

Enable one shot mode

Enable this parameter to have the DMA channel complete an entire *transfer* in response to an interrupt event trigger.

This option allows a single DMA channel and peripheral to dominate resources, and may streamline processing, but it also creates the potential for resource conflicts and delays.

Disable this parameter to have DMA complete one *burst* per channel per interrupt.

Sync enable

When **Interrupt source** is set to SEQ1INT, enable this parameter to reset the DMA wrap counter when it receives the ADCSYNC signal from SEQ1INT. This way, the wrap counter and the ADC channels remain synchronized with each other.

If **Interrupt source** is not set to SEQ1INT, **Sync enable** does not alter the results.

This parameter corresponds to bit 12 (SYNCE) of the Mode Register (MODE).

Enable continuous mode

Select this parameter to leave the DMA channel enabled upon completing a transfer. The channel will wait for the next interrupt event trigger.

Clear this parameter to disable the DMA channel upon completing a transfer. The DMA module disables the DMA channel by clearing the RUNSTS bit in the CONTROL register when it completes the transfer. To use the channel again, first reset the RUN bit in the CONTROL register.

Enable DST sync mode

When **Sync enable** is enabled, enabling this parameter resets the destination wrap counter (DST_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

Disabling this parameter resets the source wrap counter (SCR_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

This parameter is associated with bit 13 (SYNCSEL) in the Mode Register (MODE).

Set channel 1 to highest priority

This parameter is only available for DMA_ch1.

Enable this setting when DMA channel 1 is configured to handle high-bandwidth data, such as ADC data, and the other DMA channels are configured to handle lower-priority data.

When enabled, the DMA module services each enabled channel sequentially until it receives a trigger from channel 1.

Upon receiving the trigger, DMA interrupts its service to the current channel at the end of the current word, services the channel 1 burst that generated the trigger, and then continues servicing the current channel at the beginning of the next word.

Disable this channel to give each DMA channel equal priority, or if DMA channel 1 is the only enabled channel.

When disabled, the DMA module services each enabled channel sequentially.

This parameter corresponds to bit 0 (CH1PRIORITY) in the Priority Control Register 1 (PRIORITYCTRL1).

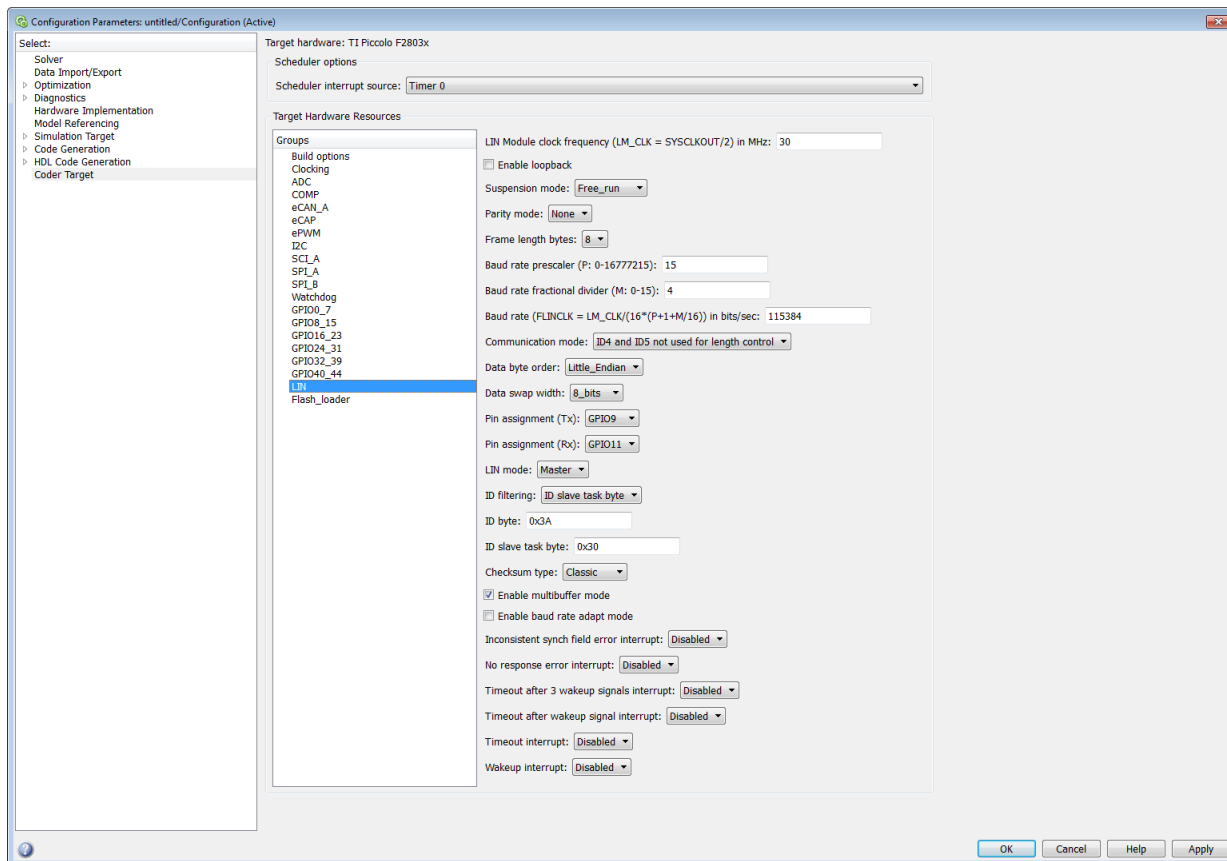
Enable overflow interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via PIE if the DMA module receives a peripheral interrupt while a previous interrupt from the same peripheral is waiting to be serviced.

This parameter is typically used for debugging during the development phase of a project.

The **Enable overflow interrupt** parameter corresponds to bit 7 (OVRINTE) of the Mode Register (MODE), and involves the Overflow Flag Bit (OVRFLG) and Peripheral Interrupt Trigger Flag Bit (PERINTFLG).

LIN



For detailed information on the LIN module, see *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2, available at the Texas Instruments Web site.

The following options configure all LIN Transmit and LIN Receive blocks within a model.

LIN Module clock frequency (LM_CLK = SYSCLKOUT/2) in MHz

Displays the frequency of the LIN module clock in MHz.

Enable loopback

To enable LIN loopback testing, select this option. While this option is enabled, the LIN module does the following:

- Internally redirects the LINTX output to the LINRX input.
- Puts the external LINTX pin into high state.
- Puts the external LINRX pin into a high impedance state.

The default is disabled (unchecked).

Suspension mode

Use this option to configure how the LIN state machine behaves while you debug the program on an emulator. If you select **Hard_abort**, entering LIN debug mode halts the transmissions and counters.

The transmissions and counters resume when you exit LIN debug mode. If you select **Free_run**, entering LIN debug mode allows the current transmit and receive functions to complete.

The default is **Free_run**.

Parity mode

Use this option to configure parity checking:

- To disable parity checking, select **None**.
- To enable odd parity checking, select **Odd**.
- To enable even parity checking, select **Even**.

The default is **None**.

In order for **ID parity error interrupt** in the LIN Receive block to generate interrupts, also enable **Parity mode**.

Frame length bytes

Set the number of data bytes in the response field, from 1 to 8 bytes.

The default is 8 bytes.

Baud rate prescaler (P: 0-16777215)

To set the LIN baud rate manually, enter a prescaler value, from 0 to 16777215. Click **Apply** to update the **Baud rate** display.

The default is 15.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

Baud rate fractional divider (M: 0–15)

To set the LIN baud rate manually, enter a fractional divider value, from 0 to 15. Click **Apply** to update the **Baud rate** display.

The default is 4.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

Baud rate (FLINCLK = $LM_CLK/(16*(P+1+M/16))$ in bits/sec

This field displays the baud rate. For more information, see “Setting the LIN baud rate”.

Communication mode

Enable or disable the LIN module from using the ID-field bits ID4 and ID5 for length control.

The default is ID4 and ID5 not used for length control

Data byte order

Set the “endianness” of the LIN message data bytes to `Little_Endian` or `Big_Endian`.

The default is `Little_Endian`.

Data swap width

Select `8_bits` or `16_bits`. If you set **Data byte order** to `Big_Endian`, the only available option for **Data swap width** is `8_bits`.

Pin assignment (Tx)

Map the LINTX output to a specific GPIO pin.

The default is GPIO9.

Pin assignment (Rx)

Map the LINRX input to a specific GPIO pin.

The default is GPIO11.

LIN mode

Put the LIN module in `Master` or `Slave` mode. The default is `Slave`.

In master mode, the LIN node can transmit queries and commands to slaves. In slave mode, the LIN module responds to queries or commands from a master node.

This option corresponds to the CLK_MASTER field in the SCI Global Control Register (SCIGCR1).

ID filtering

Select which type of mask filtering comparison the LIN module performs, **ID byte** or **ID slave task byte**.

If you select **ID byte**, the module uses the RECID and ID-BYTE fields in the LINID register to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module does not report matches.

If you select **ID slave task**, the module uses the RECID and ID-SlaveTask byte to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module reports matches.

The default is **ID slave task byte**.

ID byte

If you set **ID filtering** to **ID byte**, use this option to set the ID BYTE, also known as the “LIN mode message ID”.

In master mode, the CPU writes this value to initiate a header transmission. In slave mode, the LIN module uses this value to perform message filtering.

The default is 0x3A.

ID slave task byte

If you set **ID filtering** to **ID slave task byte**, use this option to set the ID-SlaveTask BYTE. The LIN node compares this byte with the Received ID and determines whether to send a transmit or receive response.

The default is 0x30.

Checksum type

Use this option to select the type of checksum. If you select **Classic**, the LIN node generates the checksum field from the data fields in the response.

If you select **Enhance**, the LIN node generates the checksum field from both the ID field in the header and data fields in the response. LIN 1.3 supports classic checksums only. LIN 2.0 supports both classic and enhanced checksums.

The default is **Classic**.

Enable multibuffer mode

When you enable (select) this checkbox, the LIN node uses transmit and receive buffers instead of just one register. This setting affects various other LIN registers, such as: checksums, framing errors, transmitter empty flags, receiver ready flags, transmitter ready flags.

The default is enabled (checked).

Enable baud rate adapt mode

The dialog box displays this option when you set **LIN mode** to **Slave**.

If you enable this option, the slave node automatically adjusts its baud rate to match that of the master node. For this feature to work, first set the **Baud rate prescaler** and **Baud rate fractional divider**.

If you disable this option, the LIN module sets a static baud rate based on the **Baud rate prescaler** and **Baud rate fractional divider**.

The default is disabled (unchecked).

Inconsistent synch field error interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

If you enable this option, the slave node generates interrupts when it detects irregularities in the synch field. This option is only relevant if you enable **Enable adapt mode**.

The default is **Disabled**.

No response error interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

If you enable this option, the LIN module generates an interrupt if it does not receive a complete response from the master node within a timeout period.

The default is **Disabled**.

Timeout after 3 wakeup signals interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When enabled, the slave node generates an interrupt when it sends three wakeup signals to the master node and does not receive a header in response. (The slave waits 1.5 seconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is having a problem recovering from low-power or sleep mode.

The default is **Disabled**.

Timeout after wakeup signal interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When enabled, the slave node generates an interrupt when it sends a wakeup signal to the master node and does not receive a header in response. (The slave waits 150 milliseconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is delayed recovering from low-power or sleep mode.

The default is **Disabled**.

Timeout interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When enabled, the slave node generates an interrupt after 4 seconds of inactivity on the LIN bus.

The default is **Disabled**.

Wakeup interrupt

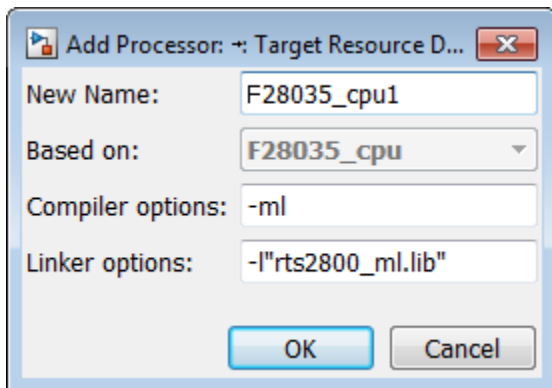
The dialog box displays this option when you set **LIN mode** to **Slave**.

When you enable this option:

- In low-power mode, a LIN slave node generates a wakeup interrupt when it detects the falling edge of a wake-up pulse or a low level on the LINRX pin.
- A LIN slave node that is “awake” generates a wakeup interrupt if it receives a request to enter low-power mode while it is receiving.
- A LIN slave node that is “awake” does not generate a wakeup interrupt if it receives a wakeup pulse.

The default is Disabled.

Add Processor Dialog Box



To add a new processor to the drop down list for the **Processors** option, click the **Add new** button on the **Board** pane. The software opens the **Add Processor** dialog box.

Note You can use this feature to create duplicates of existing processors with minor changes to the compiler and linker options. Avoid using this feature to create profiles for processors that are not already supported.

New Name

Provide a name to identify your new processor. Use a valid C string. The name you enter in this field appears on the list of processors after you add the new processor.

If you do not provide an entry for each parameter, the coder product returns an error message without creating a processor entry.

Based On

When you add a processor, the dialog box uses the settings from the currently selected processor as the basis for the new one. This parameter displays the currently selected processor.

Compiler options

Identifies the processor family of the new processor to the compiler. The string depends on the processor family or class.

For example, to set the compiler switch for a new C5509 processor, enter `-m1`. The following table shows the compiler switch string for supported processor families.

Processor Family	Compiler Switch String
C62xx	
C64xx	
C67xx	
DM64x and DM64xx	
C55xx	-m1
C28xx, F28xx, R28xx, F28xxx	-m1

Linker options

You can use this parameter to specify linker command options. The IDE uses these options to modify how it links project files when you build a project. To get information about specific linker options you can enter here, consult the documentation for your IDE.

Target Hardware Resources Tab: Linux, VxWorks, or Windows

This tab appears when both of the following conditions are true:

- The **IDE/Tool Chain** parameter is set to a toolchain that builds generated code to run on Windows, Linux, or VxWorks (requires an Embedded Coder license).
- The **Operating System** parameter on the Board tab is set to Windows, Linux, or VxWorks

Scheduling Mode

When you select `free-running`, the model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.

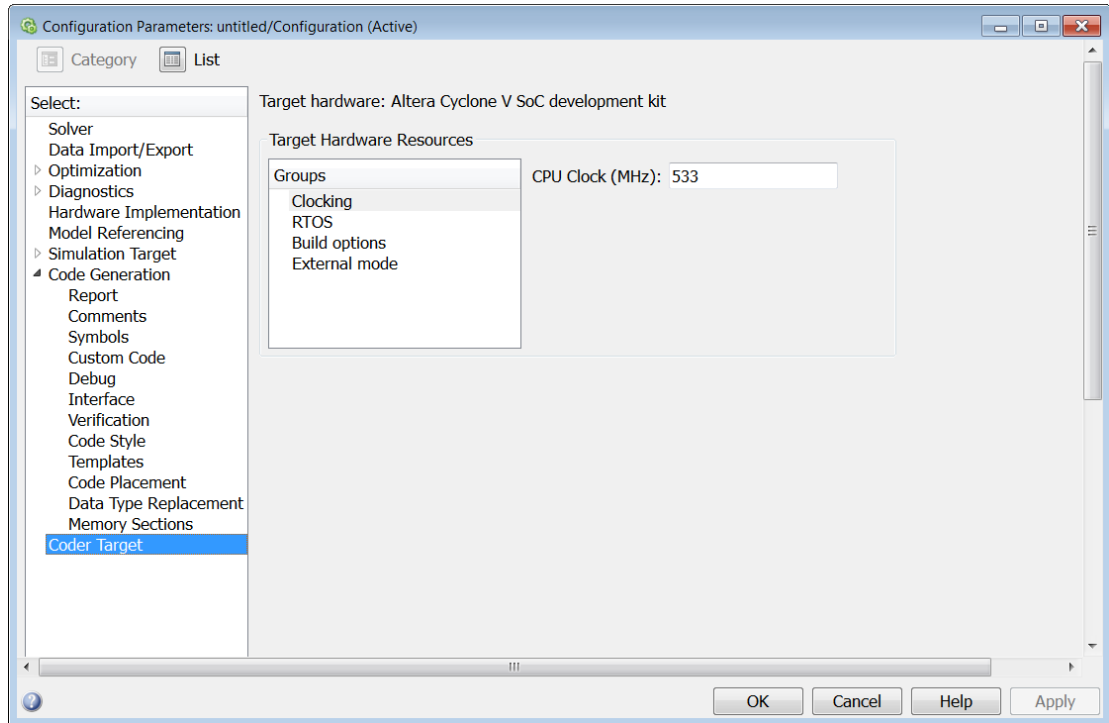
When you select `real-time`, the model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a real-time clock.

Base rate task priority

The base rate in the model maps to a thread and runs as fast as possible. You can use the value of the base rate priority to set a static priority for the base rate task. By default, this rate is 40.

This parameter is not available for Windows.

Coder Target Pane: Altera Cyclone V SoC development kit, Arrow SoCKit development board



In this section...

“Coder Target Pane Overview” on page 3-177

“Coder Target” on page 3-178

“Clocking” on page 3-178

“RTOS” on page 3-179

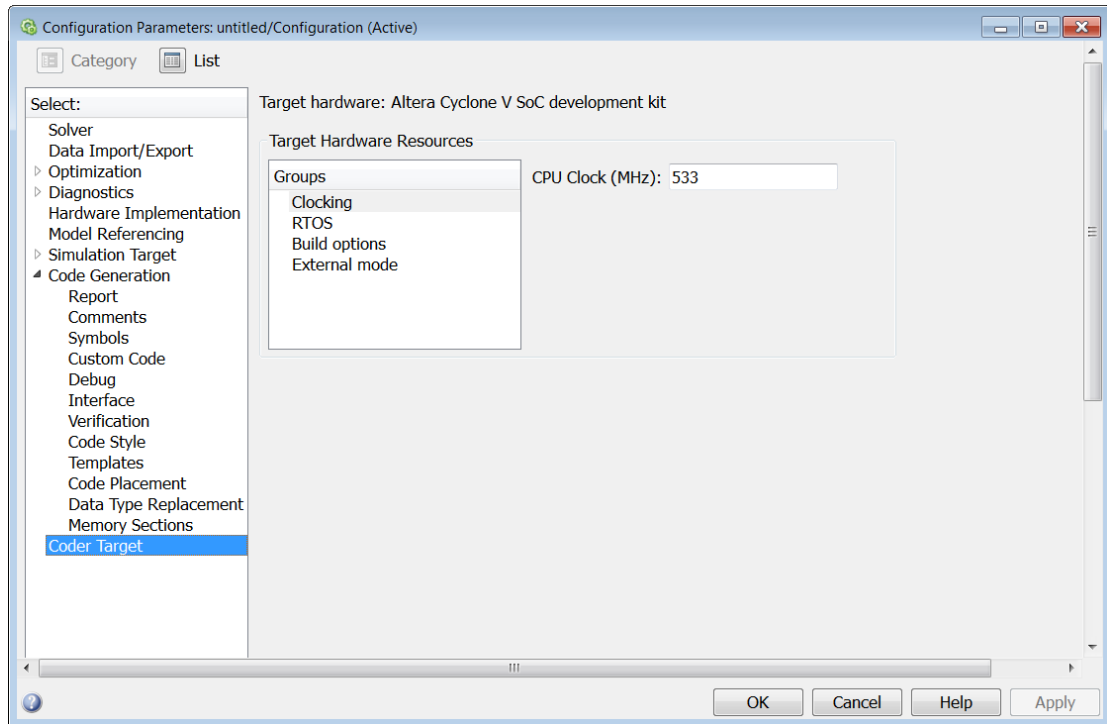
“Build Options” on page 3-179

“External mode” on page 3-180

Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

Coder Target



To configure the Coder Target pane for Altera® SoC hardware, go to the Code Generation pane. Then,

- Set **System target file** to `ert.tlc`
- Set **Target hardware** to Altera Cyclone V SoC development kit or Arrow SoCKit development board

Clocking

CPU Clock (MHz)

Specify the CPU clock frequency of the real ARM® Cortex® processor on the target hardware.

RTOS

Target RTOS

This parameter sets the static priority of the base rate task. By default, the priority is 40.

Base Rate Task Priority

This parameter sets the static priority of the base rate task. By default, the priority is 40.

Build Options

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Build, load and run

Build, load and run

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.
- 3 Loads the executable and libraries into the QEMU emulator.
- 4 Runs the executable in the QEMU emulator.

Build

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

Command-Line Information

Parameter: buildAction

Type: string

Value: Build | Build_load_and_run |

Default: Build_load_and_run

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

External mode

Communication interface

Select the transport layer External mode uses to exchange data between the host computer and the target hardware.

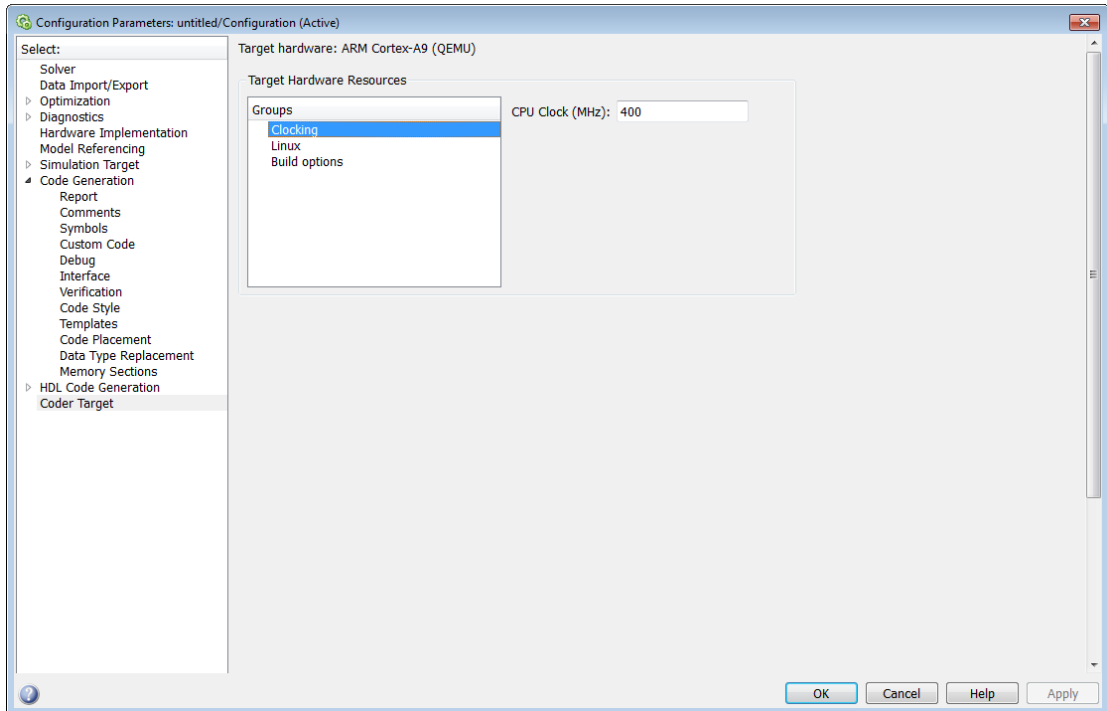
IP address

Specify the IP address of the target hardware. By default, this value is an environment variable, \$(SOCFPGA_IPADDRESS), which reuses the IP address from the most recent connection to the target hardware.

Verbose

Display verbose messages in the MATLAB Command Window.

Coder Target Pane: ARM Cortex-A9 (QEMU)



In this section...

“Coder Target Pane Overview” on page 3-181

“Coder Target” on page 3-182

“Clocking” on page 3-182

“Linux” on page 3-183

“Build Options” on page 3-183

“External mode” on page 3-184

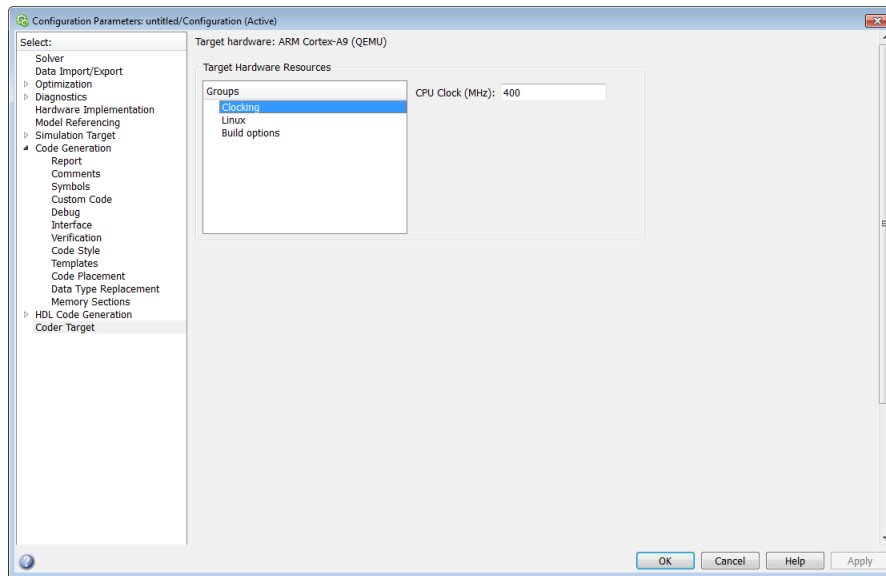
Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

See Also

Coder Target: Target Hardware Resources Tab Overview

Coder Target



The Coder Target pane is visible when the following parameters are on the Code Generation pane are both set as follows:

- **System target file** is `ert.tlc`
- **Target hardware** is ARM Cortex-A9 (QEMU)

Changing the **Target hardware** parameter changes the number and type of parameters that are available on the Coder Target pane.

Clocking

CPU Clock (MHz)

Enter the actual CPU clock frequency in MHz. This value does not set the processor frequency.

The software uses this value to calculate the speed of the processor.

Linux

Base Rate Task Priority

This parameter sets the static priority of the base rate task. By default, the priority is 40.

Build Options

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Build, load, and run

Build, load, and run

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.
- 3 Loads the executable and libraries into the QEMU emulator.
- 4 Runs the executable in the QEMU emulator.

Build

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

Command-Line Information

Parameter: buildAction

Type: string

Value: Build | Build_load_and_run |

Default: Build_load_and_run

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

External mode

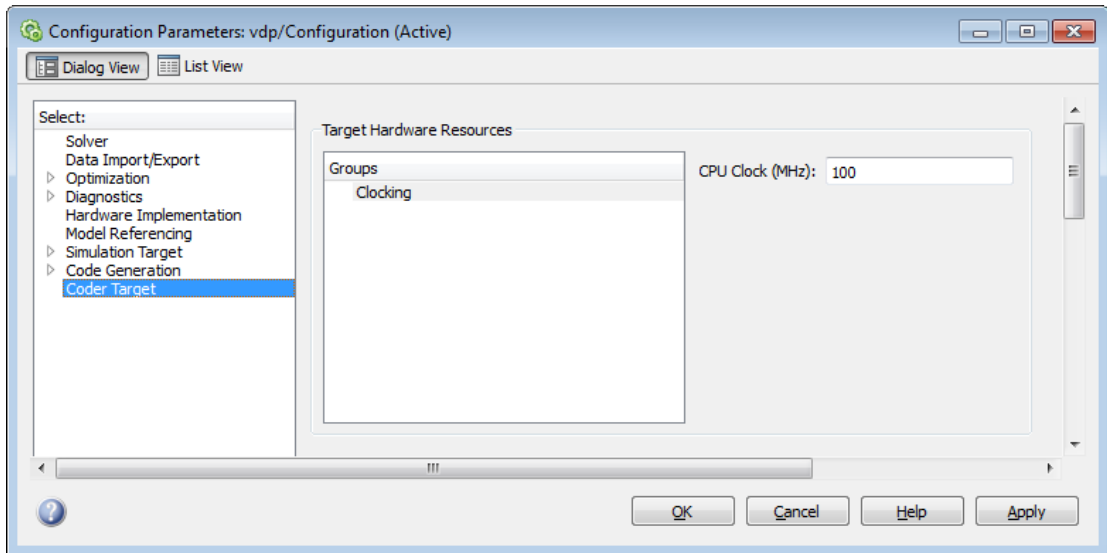
Communication interface

Select the transport layer External mode uses to exchange data between the host computer and the target hardware.

Verbose

Display verbose messages in the MATLAB Command Window.

Coder Target Pane: ARM Cortex-M3 (QEMU)



In this section...

- “Coder Target Pane Overview” on page 3-185
- “Coder Target” on page 3-185
- “Clocking” on page 3-186
- “External mode” on page 3-184

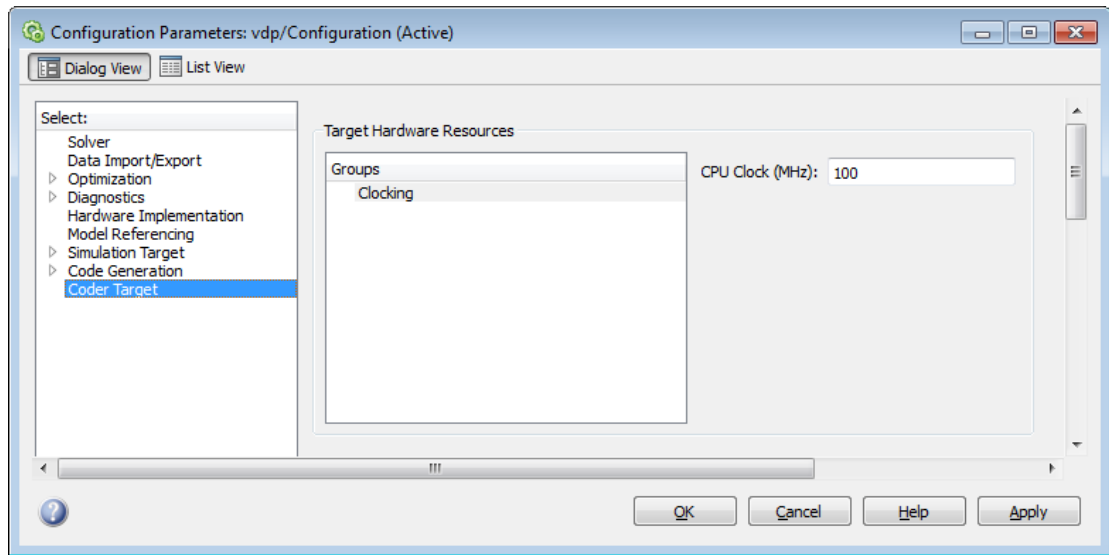
Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

See Also

Coder Target: Target Hardware Resources Tab Overview

Coder Target



The Coder Target pane is visible when the following parameters are on the Code Generation pane are both set as follows:

- **System target file** is `ert.tlc`
- **Target hardware** is ARM Cortex-M3 (QEMU)

Changing the **Target hardware** parameter changes the number and type of parameters that are available on the Coder Target pane.

Clocking

CPU Clock (MHz)

Specify the CPU clock frequency of a real ARM Cortex-M3 processor in MHz. The QEMU uses this value to emulate an ARM Cortex-M3 processor.

External mode

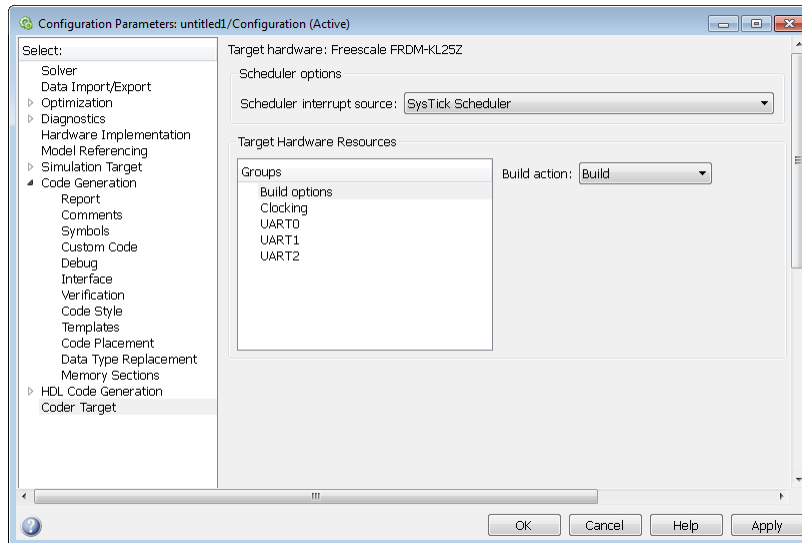
Communication interface

Select the transport layer External mode uses to exchange data between the host computer and the target hardware.

Verbose

Display verbose messages in the MATLAB Command Window.

Coder Target Pane: Embedded Coder Support Package for Freescale FRDM-KL25Z Board



In this section...

“Coder Target Pane Overview” on page 3-188

“Coder Target” on page 3-189

“Scheduler options” on page 3-189

“Build Options” on page 3-189

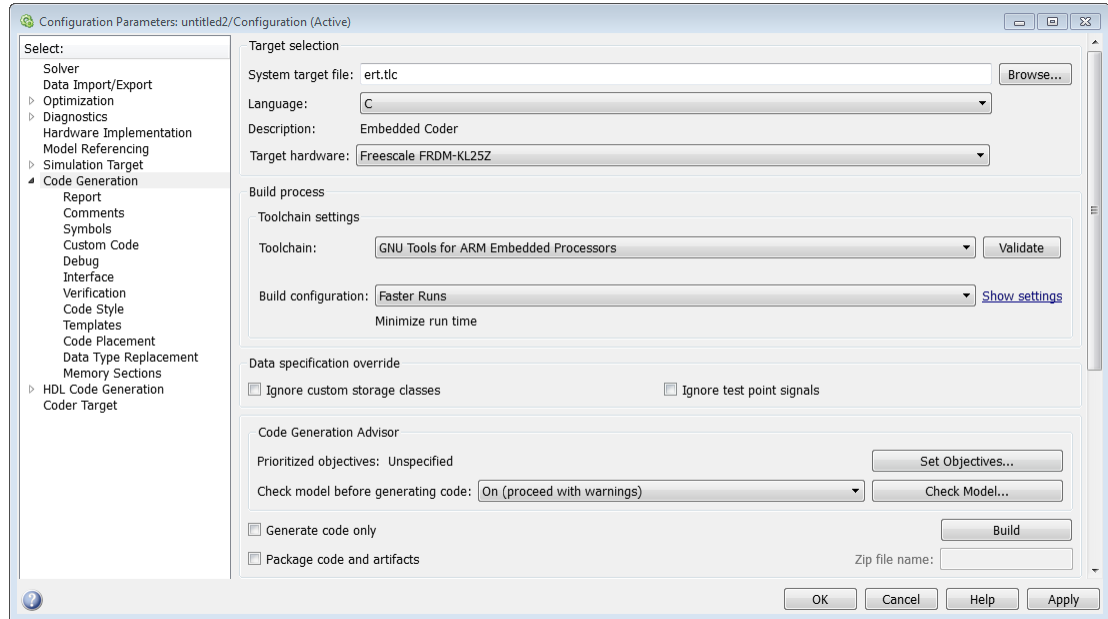
“Clocking” on page 3-190

“UART0, UART1, and UART2” on page 3-191

Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

Coder Target



This Coder Target pane is visible when the following parameters on the Code Generation pane are both set as follows:

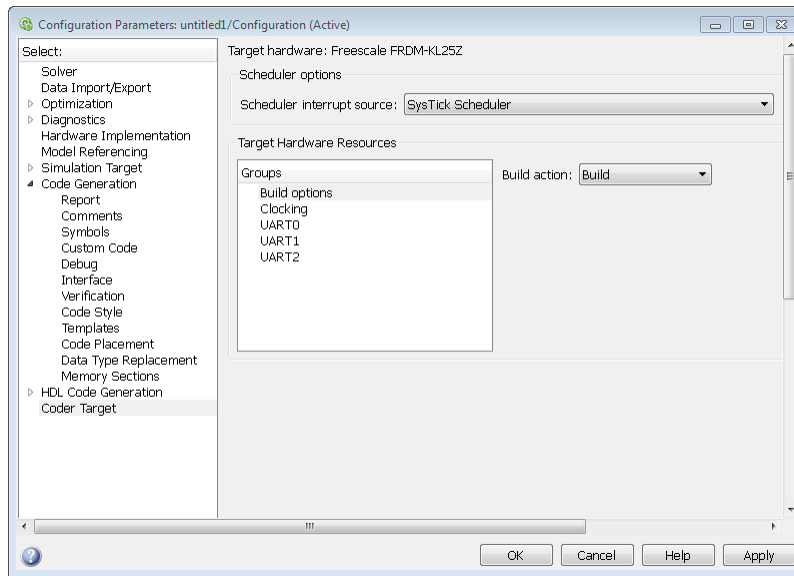
- **System target file** is `ert.tlc`
- **Target hardware** is `Freescale FRDM-KL25Z`

Scheduler options

Scheduler interrupt source

Select the source of the scheduler interrupt.

Build Options

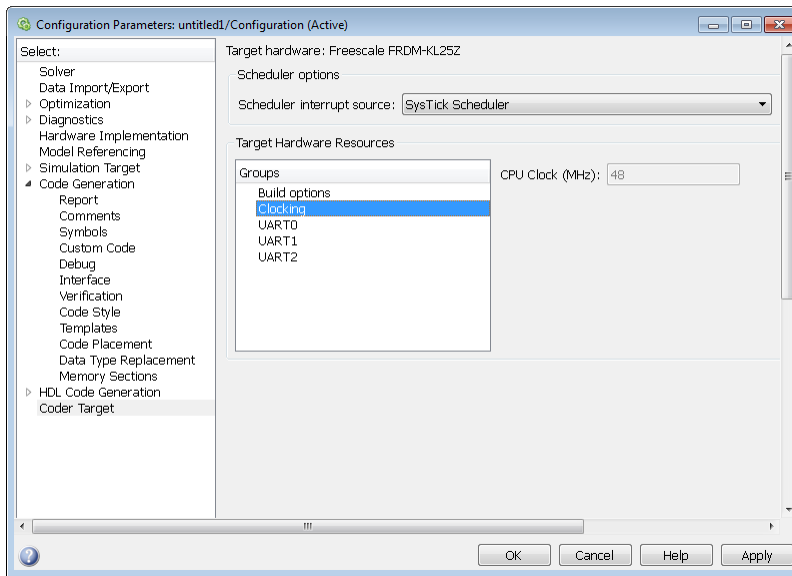


Select build options to specify how the build process should take place during code generation.

Build action

Select an option to specify that you want only build or build, load, and run actions during the build process.

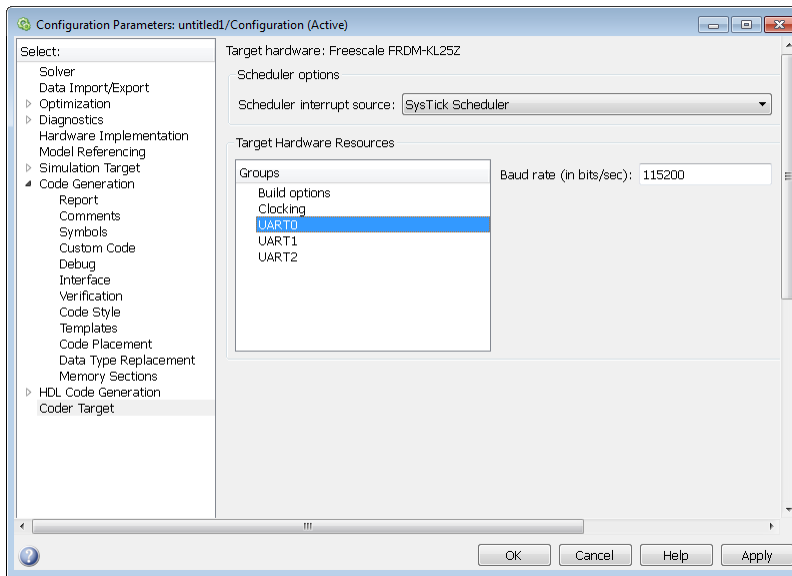
Clocking



CPU Clock (MHz)

This is the CPU clock frequency of the Freescale FRDM-KL25Z processor on the target hardware.

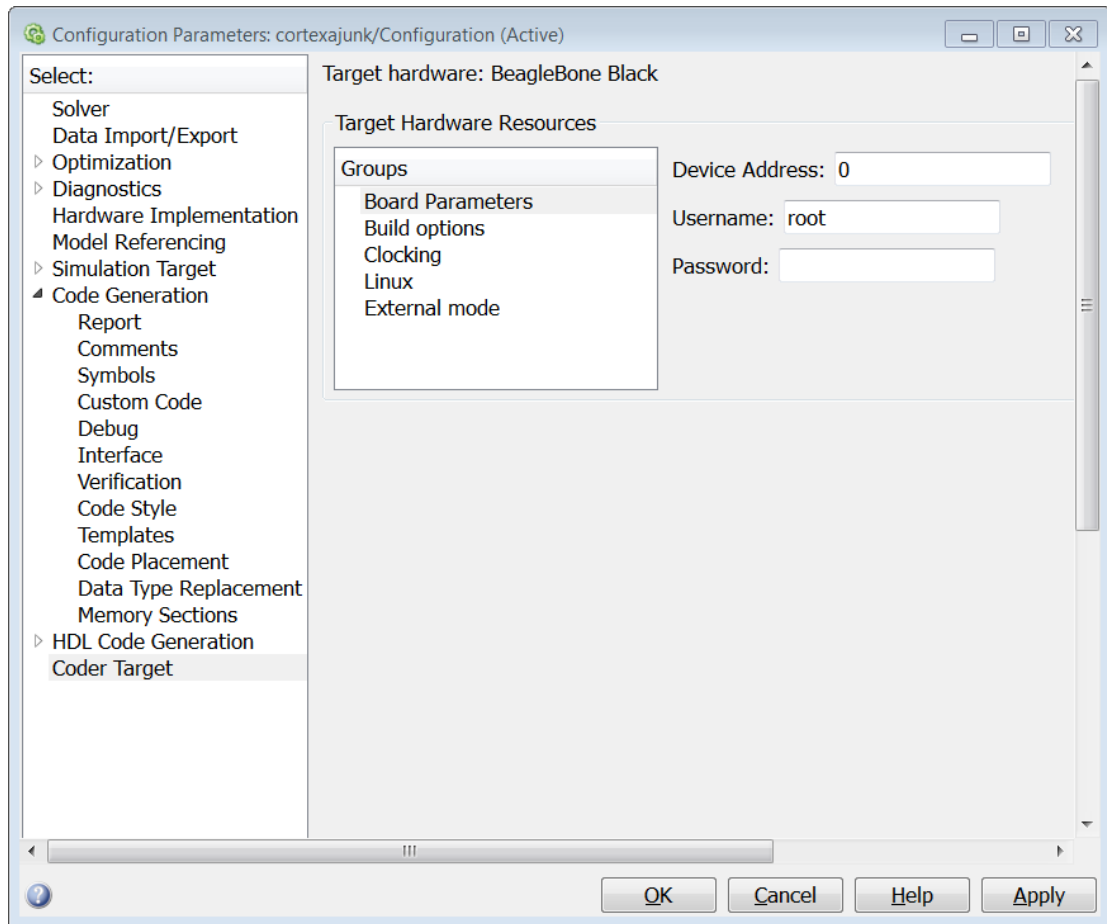
UART0, UART1, and UART2



Baud rate (in bits/sec)

Specify the baud rate for UARTx serial interfaces.

Coder Target Pane: BeagleBone Black Hardware



In this section...

“Coder Target Pane Overview” on page 3-194

“Coder Target” on page 3-194

“Board Parameters” on page 3-195

“Build Options” on page 3-183

“Clocking” on page 3-182

In this section...

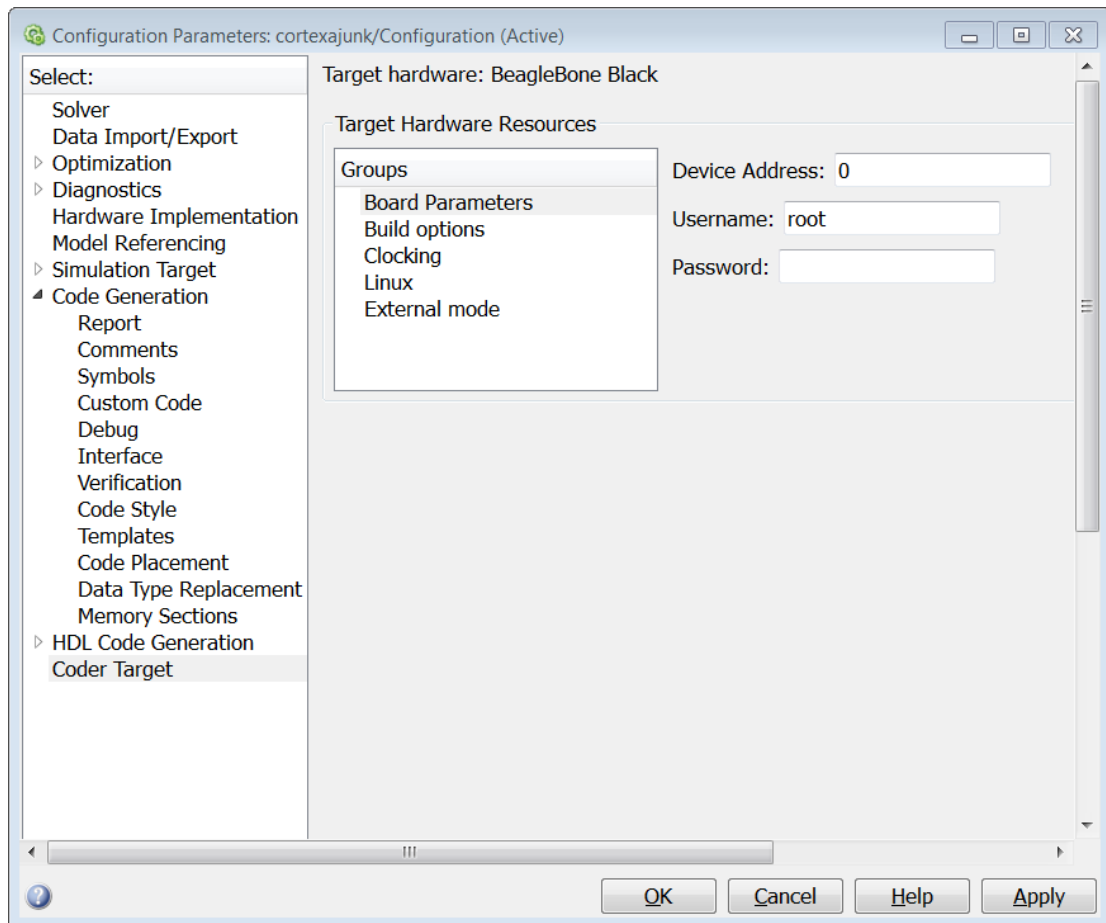
“Linux” on page 3-183

“External mode” on page 3-184

Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

Coder Target



This Coder Target pane is visible when the following parameters are on the Code Generation pane are both set as follows:

- **System target file** is `ert.tlc`
- **Target hardware** is `BeagleBone Black Hardware`

Board Parameters

Device Address

Specify the device address.

Username

Specify the user name of the root user. By default, this value is `root`.

Password

Specify the password of the root user.

Build Options

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Build, load, and run

Build, load, and run

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.
- 3 Loads the executable and libraries into the QEMU emulator.
- 4 Runs the executable in the QEMU emulator.

Build

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.

- 2 Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

Command-Line Information

Parameter: buildAction

Type: string

Value: Build | Build_load_and_run |

Default: Build_load_and_run

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

Clocking

CPU Clock (MHz)

Enter the actual CPU clock frequency in MHz. This value does not set the processor frequency.

The software uses this value to calculate the speed of the processor.

Linux

Base Rate Task Priority

This parameter sets the static priority of the base rate task. By default, the priority is 40.

External mode

Communication interface

Select the transport layer External mode uses to exchange data between the host computer and the target hardware.

Verbose

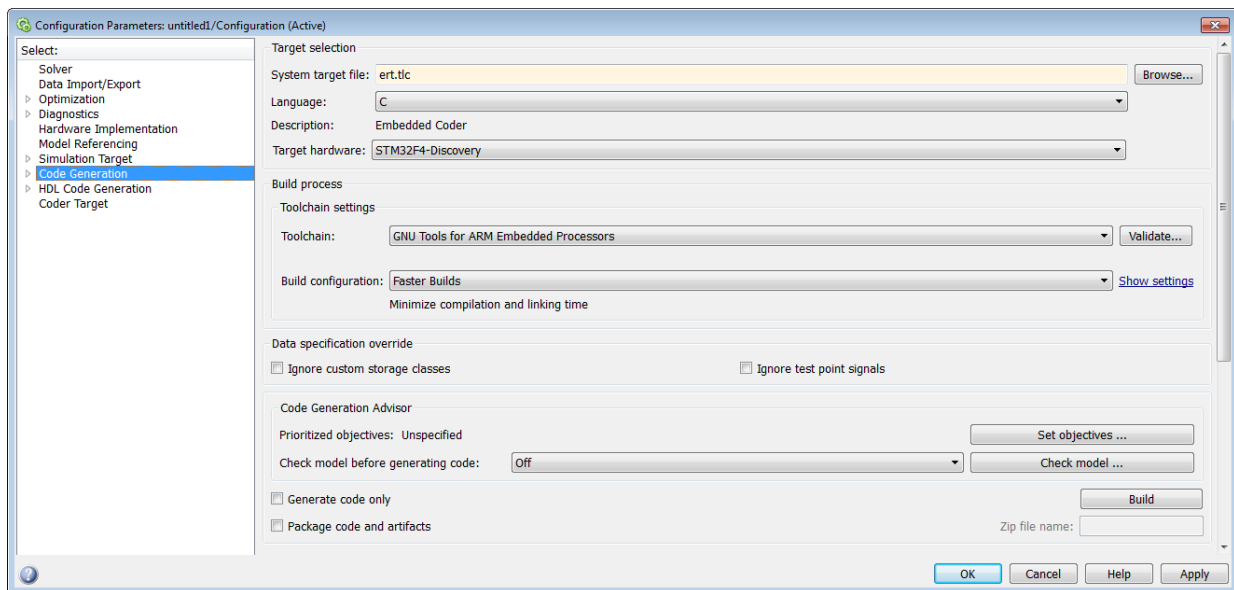
Display verbose messages in the MATLAB Command Window.

Coder Target Pane: Support Package for STMicroelectronics STM32F4 Discovery Hardware

The Coder Target pane is visible when you set both of the following parameters on the Code Generation pane as follows:

- **System target file** is `ert.tlc`
- **Target hardware** is STM32F4–Discovery hardware (not None)

Changing the **Target hardware** parameter changes the number and type of parameters that are available on the Coder Target pane.



In this section...

“Coder Target Pane: STM32F4–Discovery Overview” on page 3-199

“Support Package for STMicroelectronics STM32F4 Discovery Hardware Settings” on page 3-199

“Scheduler options” on page 3-200

“Build options” on page 3-201

“Clocking” on page 3-201

In this section...
“PIL” on page 3-202
“ADC Common” on page 3-203
“ADC 1, ADC 2, ADC 3” on page 3-205
“GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I” on page 3-206
“Coder Target Pane” on page 3-207
“System target file” on page 3-208
“Target hardware” on page 3-209
“Toolchain” on page 3-209

Coder Target Pane: STM32F4–Discovery Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

See Also

- “Coder Target Pane” on page 3-207
- “Coder Target Pane: Support Package for STMicroelectronics STM32F4 Discovery Hardware” on page 3-198

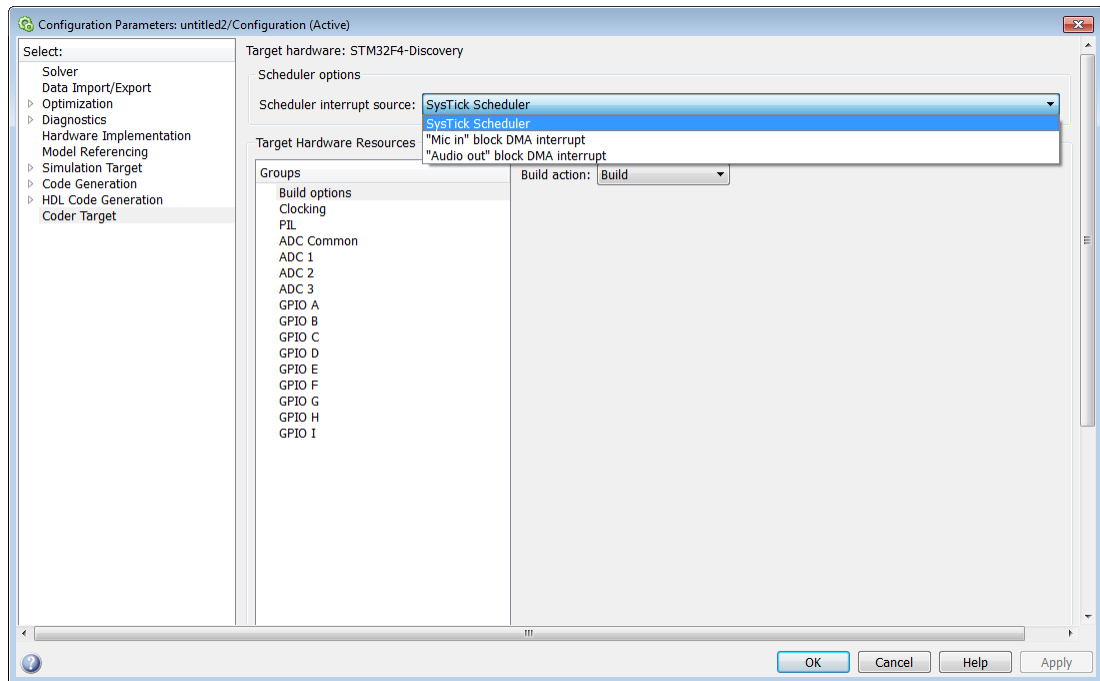
Support Package for STMicroelectronics STM32F4 Discovery Hardware Settings

The following table provides links to topics for Support Package for STMicroelectronics® STM32F4 Discovery hardware processors.

Peripheral Name	Description
“Scheduler options” on page 3-200	The scheduler parameter to select the scheduler interrupt source
“Build options” on page 3-201	The build options to specify how the build process should take place during code generation
“Clocking” on page 3-201	The clocking parameters to view the CPU clock rate

Peripheral Name	Description
“PIL” on page 3-202	The Processor—in—the—Loop (PIL) parameters to configure PIL communication parameters, such as interface and COM port
“ADC Common” on page 3-203	The Analog to Digital Converter (ADC) parameters to set the common ADC peripheral parameters
“ADC 1, ADC 2, ADC 3” on page 3-205	The parameters to configure different channels on the three ADCs, ADC1, ADC2, and ADC3
“GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I” on page 3-206	The GPIO parameters to configure different GPIO pins

Scheduler options



The source of the scheduler interrupt. The different scheduler options available are as follows:

Scheduler interrupt source

- **SysTick Scheduler** – The SysTick timer interrupt schedules the model at the base rate. This timer is available in the ARM Cortex-M3 and the ARM Cortex-M4 based processors. When you select STM32F4-Discovery as the target hardware, this option is the default Scheduler option.
- **“Mic in” block DMA interrupt** – The model is scheduled based on the DMA interrupt rate of the Mic in block. The Mic in block generates a DMA (DMA1-Channel0-Stream3) interrupt every 1 ms. The base rate of the model is set to 1ms, regardless of the other faster base rates in the model. Make sure that the model does not contain blocks with sample rates higher than the Mic in block.
- **“Audio out” block DMA interrupt** – The model is scheduled based on the DMA interrupt rate of the Audio out block. The DMA interrupt rate depends on the frame size that is input to the Audio out block in the model. Make sure that the model does not contain blocks with sample rates higher than the Audio out block.

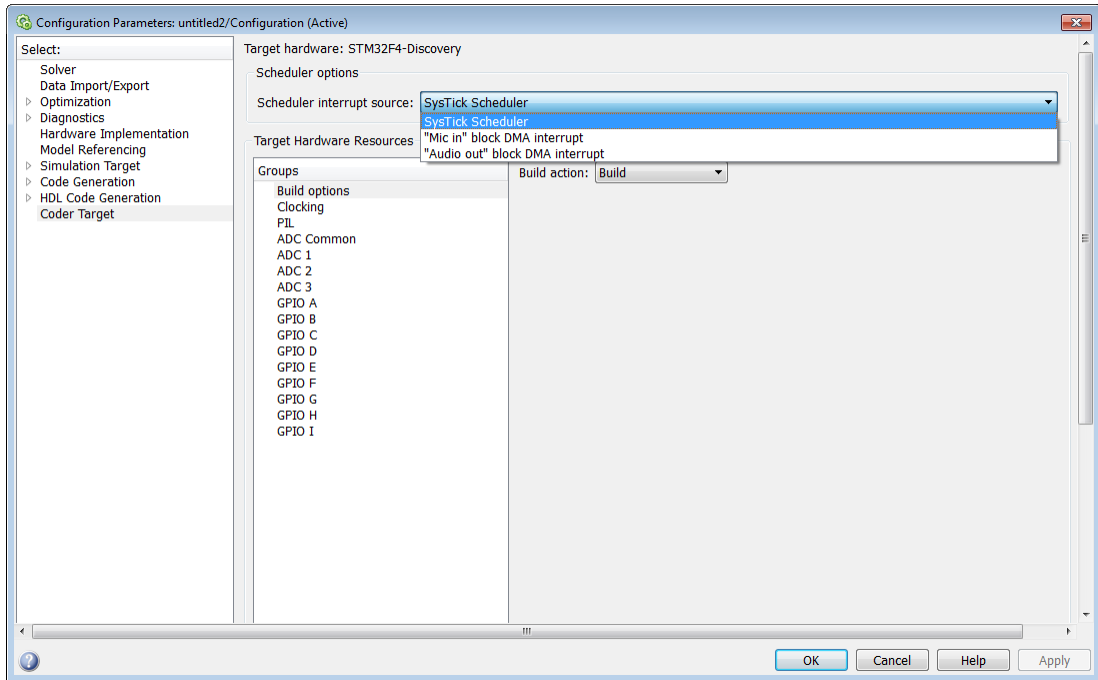
For example, if you select an audio sample frequency of 48000 Hz and you input data of 96 audio samples/frame to the right and the left channels of the Audio out block, the “Audio out” block DMA interrupt occurs every $96/48000 = 0.002$ secs (2 ms). You must schedule the model base rate at 2 ms.

Build options

Use the build options to specify how the build process should take place during code generation.

- **Build action**
is the option to specify if you want only build or build, load and run actions during code generation.

Clocking

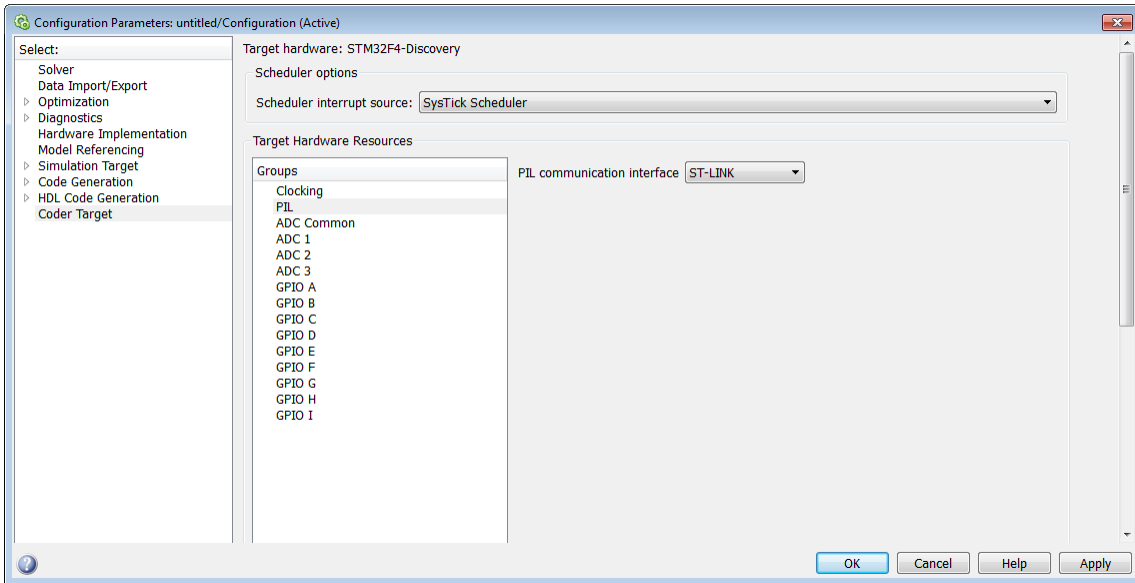


Use the clocking option to achieve the CPU Clock rate specified.

CPU Clock (MHz)

The CPU clock rate.

PIL



Use the PIL options to set PIL (Processor-in-the-Loop) communications parameters.

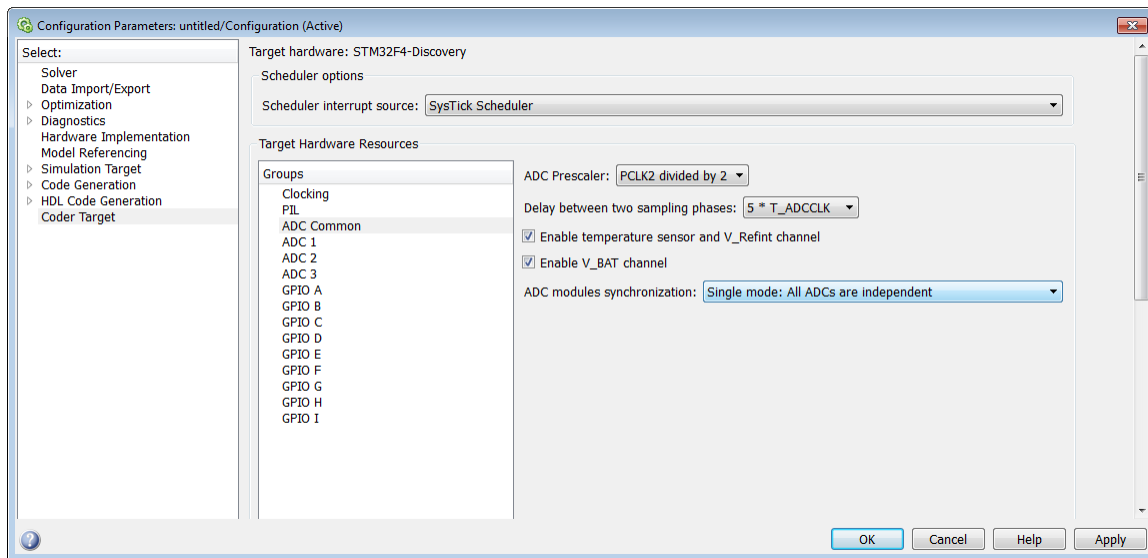
PIL communication interface

The interface used for PIL communication.

COM port

The COM port used PIL communication.

ADC Common



Use the ADC Common options to set the common ADC parameters.

ADC Prescaler

The option to select the PCLK divider.

Delay between two sampling phases

The option to select the time delay between two sampling phases.

Enable temperature sensor and V_Refint channel

The option to enable the temperature sensor and the V_Refint channel.

Enable V_BAT channel

The option to enable the V_BAT channel.

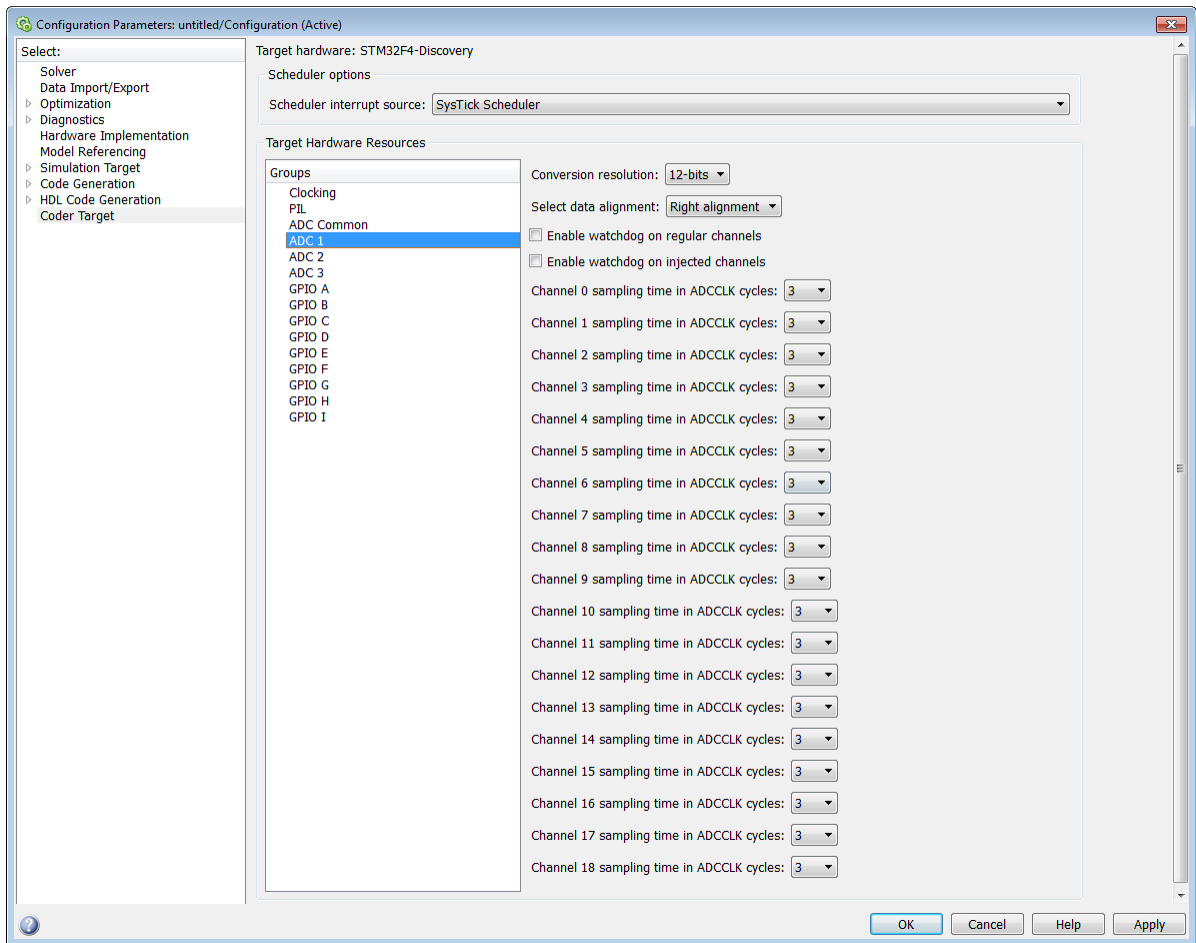
ADC modules synchronization

The option that you select for ADC module synchronization. The different options you available are:

- **Single mode: All ADCs are independent** — Select this option when the three ADCs must perform the conversion independently.
- **Dual mode: ADC1 and ADC2 combined, ADC3 independent** — Select this option when you want to combine ADC1 and ADC2 and ADC3 to perform conversion independently.

- **Triple mode: ADC1, ADC2, and ADC3 combined** — Select this option when you want to combine ADC1, ADC2, and ADC3 together for conversion.

ADC 1, ADC 2, ADC 3



Use the three ADC1, ADC2, and ADC3 parameters to configure the sample time in ADDCLK cycles for different channels.

Conversion resolution

The resolution that you select for conversion.

Convert number of channels in discontinuous mode

The number of channels to convert in discontinuous mode.

Select data alignment

The option that you select for the alignment of data after conversion. The data can be right or left aligned.

Enable watchdog on regular channels

The option you select to enable the watchdog on regular channels.

Enable watchdog on injected channels

The option you select to enable the watchdog on injected channels.

Enable discontinuous conversion on injected group

The option that you select to enable the discontinuous conversion on injected group.

Enable discontinuous conversion on regular group

The option that you select to enable discontinuous conversion on regular group.

DMA mode for multi ADC mode

The option that you select for DMA mode in multi ADC mode.

Enable DMA selection for multi ADC mode

The option that you select to enable DMA selection for multi ADC mode.

Multi ADC mode selection

The option that you select for multi ADC mode.

Watchdog on channel

The option to select channel for watchdog.

Watchdog lower threshold

The lower threshold of the watchdog.

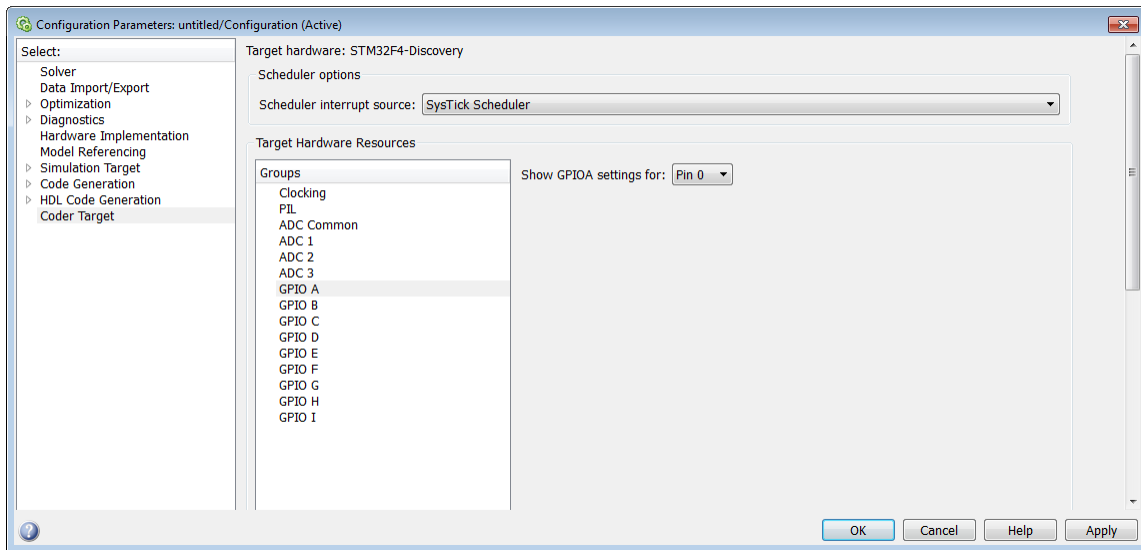
Watchdog higher threshold

The higher threshold of the watchdog.

Channel # sampling time in ADCCLK cycles

The sampling time that you select in ADCCLK cycles for channel numbers from 1 to 18.

GPIO A, GPIO B, GPIO C, GPIO D, GPIO E, GPIO F, GPIO G, GPIO H, GPIO I



Use the GPIO A-I parameters to configure the pins for input/output.

Show GPIO# settings for

The pin number that you select to show the GPIO settings.

Select output type for Pin

The output type that you select for pins 0 to 15.

Select output speed for Pin

The output speed that you select for pins 0 to 15.

Select pull mode for Pin

The pull mode that you select for pins from 0 to 15.

Coder Target Pane

The Coder Target pane is visible when you set the following parameters on the Code Generation pane as follows:

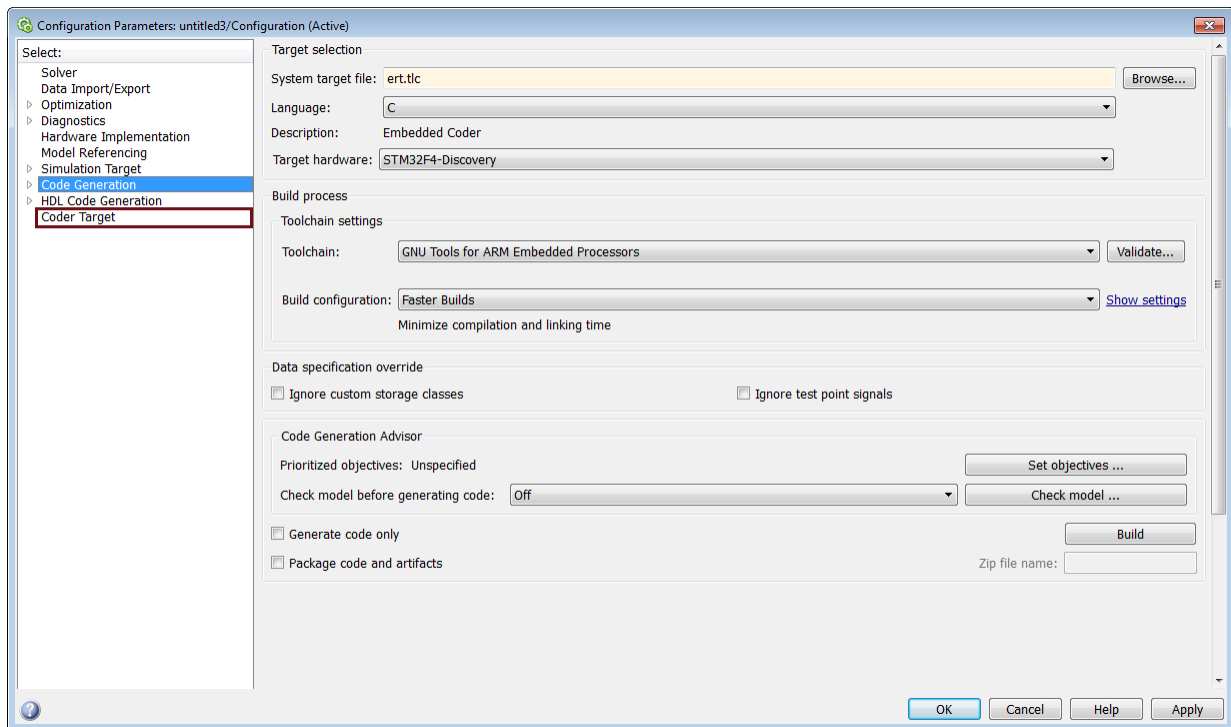
System target file to `ert.tlc`.

Target hardware to a specific type of hardware (not None).

If you are a new user, the `ert.tlc` system target file is the preferred option over the `idelink_ert.tlc`. This option provides a uniform model configuration workflow across the phases of development such as simulation, target prototyping and production code generation.

The features that the `ert.tlc` workflow supports include:

- Production code generation workflow.
- Auto-download and run (when supported with the respective vendor tools).



System target file

Specify the system target file.

Settings

Default: `grt.tlc`

Use the System Target File Browser. Click the **Browse** button, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command. For uniform model configuration workflow, select `ert.tlc`.

Target hardware

Select the target hardware for which to generate code.

The **Coder Target** pane is visible with corresponding peripherals, when you select a target hardware. Changing the **Target hardware** parameter changes the peripherals visible on the **Coder Target** pane.

Toolchain

Based on the target hardware you select, the corresponding tool chain is automatically selected. You can select a different value from the **Toolchain** list.

Note The above parameters settings are specific to **Coder Target** pane. For the other parameter settings on the screen, see “Code Generation Pane: General”.

Coder Target Pane: Texas Instruments C2000 Processors

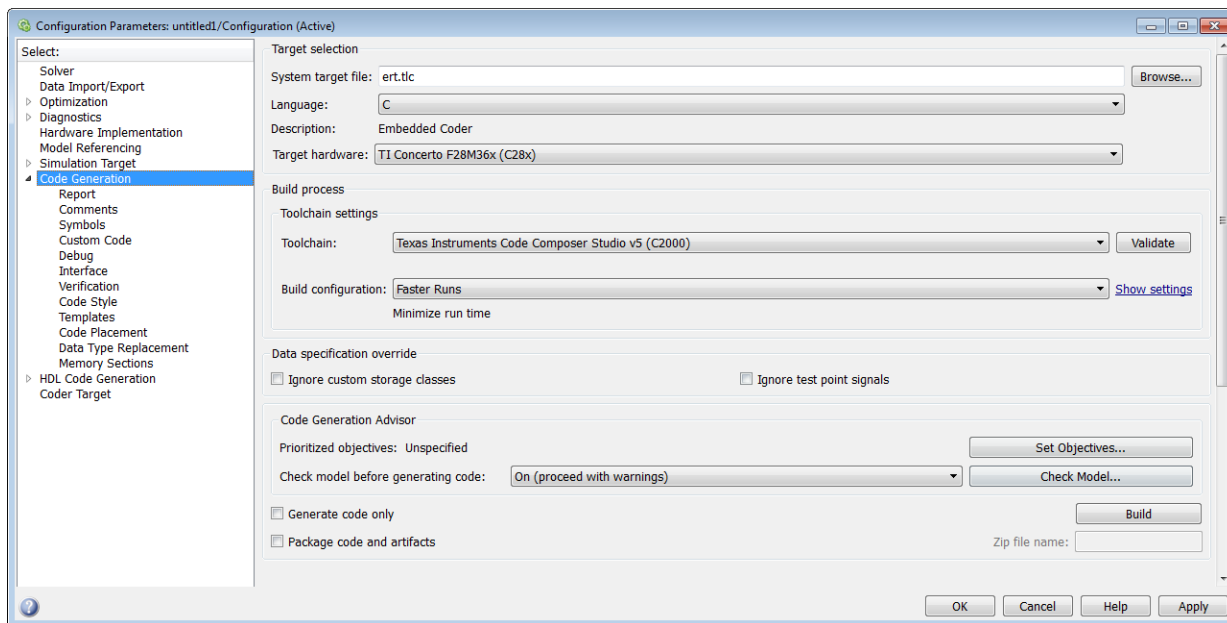
You will see the **Coder Target** pane is visible when the following parameters are on the Code Generation pane are both set as follows:

- **System target file** is `ert.tlc`
- **Target hardware** is one of the TI hardware for (not None).

Changing the **Target hardware** parameter changes the number and type of parameters that are available on the **Coder Target** pane.

If you are already using an older version of the software, you can upgrade from the existing models of `idelink_ert.tlc` or `idelink_grt.tlc` to `ert.tlc`, using the Upgrade Advisor.

Note To setup the Code Composer Studio, see “Setting Up Code Composer Studio (ert.tlc System Target File)”.



In this section...

“Coder Target Pane: TI C2000 Processors Overview” on page 3-211

“Texas Instruments C2000 Settings” on page 3-212

“Scheduler options” on page 3-213

“Build options” on page 3-214

“Clocking” on page 3-123

“ADC” on page 3-126

“COMP” on page 3-128

“eCAN_A, eCAN_B” on page 3-129

“eCAP” on page 3-131

“ePWM” on page 3-133

“I2C” on page 3-135

“SCI_A, SCI_B, SCI_C” on page 3-141

“SPI_A, SPI_B, SPI_C, SPI_D” on page 3-144

“eQEP” on page 3-147

“Watchdog” on page 3-149

“GPIO” on page 3-151

“Flash_loader” on page 3-156

“DMA_ch[#]” on page 3-158

“LIN” on page 3-167

“Coder Target Pane” on page 3-267

“System target file” on page 3-268

“Target hardware” on page 3-269

“Toolchain” on page 3-269

Coder Target Pane: TI C2000 Processors Overview

Configure the parameters for properties of the physical hardware, such as peripherals for Texas Instruments C2000.

See Also

- “Coder Target Pane: TI C2000 Processors Overview” on page 3-211

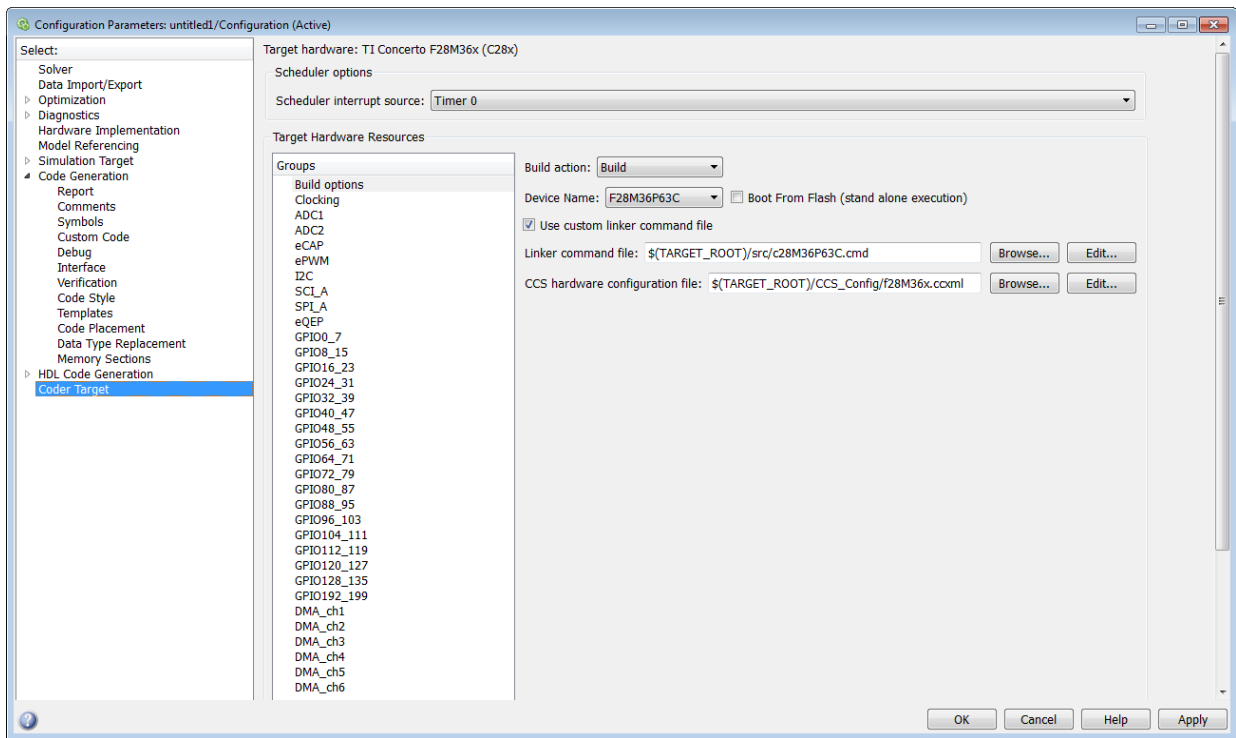
Texas Instruments C2000 Settings

The following table provides links to topics for Texas Instruments C2000 processors.

Peripheral Name	Description
	Build parameters for the build process
	Clocking parameters to adjust clock settings and match custom oscillator frequencies
	Analog-to-Digital Converter (ADC) parameters
“COMP” on page 3-128	Parameters to assign COMP pins to GPIO pins
“eCAN_A, eCAN_B” on page 3-129	Enhanced Controller Area Network (eCAN) parameters for modules A or B
	Enhanced Capture (eCAP) parameters for pin mapping to GPIO
	Enhanced Pulse Width Modulation (ePWM) parameters for pin mapping to GPIO
	Inter-Integrated Circuit (I2C) parameters for communications
	Serial Communications Interface (SCI) parameters for communications with modules A, B, or C
	Serial Peripheral Interface (SPI) parameters for communications with module A, B, C, or D
	Enhanced Quadrature Encoder Pulse (eQEP) parameters for pin mapping to GPIO
“Watchdog” on page 3-149	Watchdog enable/disable and timing
	General Purpose Input Output (GPIO) parameters for input qualification types
“Flash_loader” on page 3-156	Flash memory loader/programmer

Peripheral Name	Description
	Direct Memory Access (DMA) parameters for channels 1 to N
“LIN” on page 3-167	Local Interconnect Network (LIN) parameters for communications

Scheduler options



Scheduler interrupt source

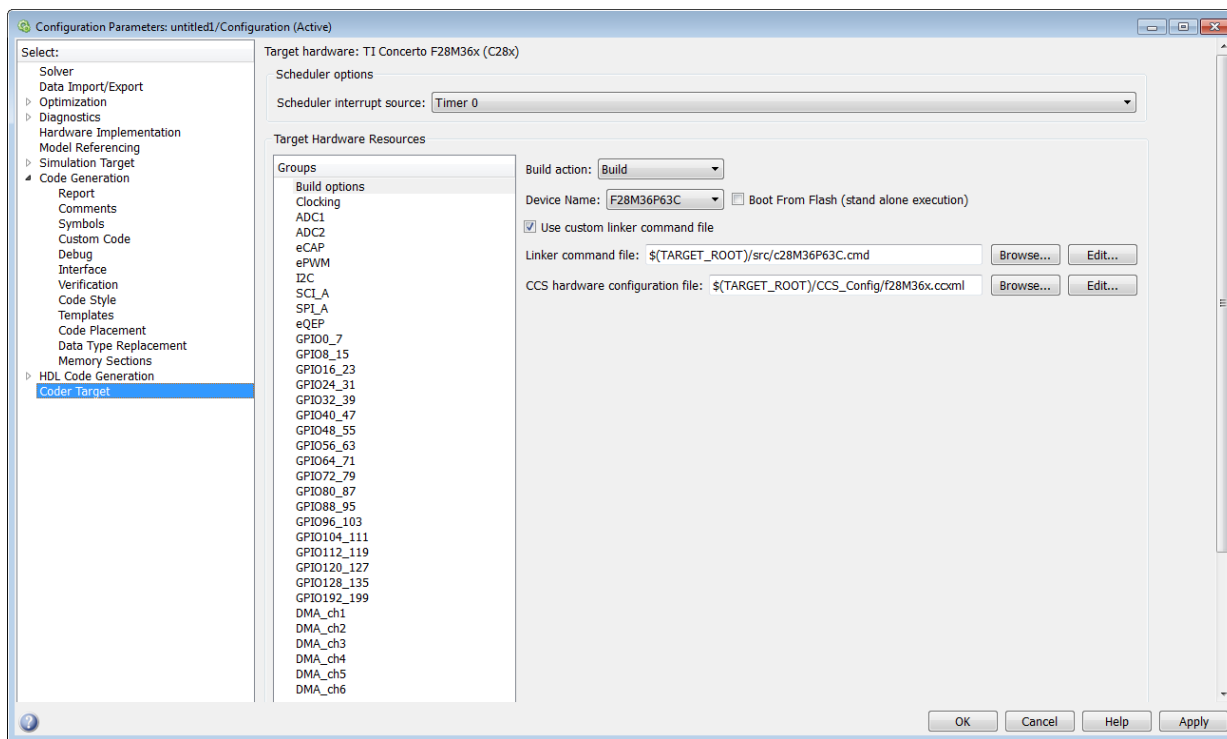
The source of the scheduler interrupt. The different scheduler options available are:

- **Timer 0** - The default option to schedule all synchronous rates present in your model with CPU Timer 0. When you select this option, the CPU Timer 0 is set to honor the base rate of the model.

- **ADCINT1** - The option to schedule all synchronous rates present in your model with ADC interrupt 1 (ADCINT1). When you select this option, make sure that ADCINT1 triggers periodically at base rate used in the model.
- **ADCINT2** - The option to schedule all synchronous rates present in your model with ADC interrupt 2 (ADCINT2). When you select this option, make sure that ADCINT2 triggers periodically at base rate used in the model.

Warning Replacing the default scheduler interrupt source Timer 0 with ADCINT1 or ADCINT2 is an advanced maneuver. It is your responsibility to ensure that you configure ADCINT1 or ADCINT2 to trigger periodically at the specified base rate. If the ADCINT1 or ADCINT2 does not trigger periodically or triggers at a different rate than the base rate, the model execution on the target is unpredictable.

Build options



Use the build options to specify how the build process should take place.

Build action

The option to specify if you want only 'build' or 'build, load, and run' action during the build process. The **build, load and run** option is supported only for TI Code Composer Studio v4/5 (C2000) tool chain.

If you select **build, load and run** option, you must provide the required CCS hardware configuration file. The TI Concerto F28M35x/ F28M36x processors support only CCS v5 and the above versions.

Device Name

The option to select a particular device from the selected processor family in the Target hardware parameter on the Code Generation pane.

Boot From Flash (stand alone execution)

The option to specify if the application has to load to the flash. If you do not select this option, the application loads to the RAM.

Use custom linker command file

The option to indicate that the custom linker command file must be used during the build action. Select this option, if you have your own custom linker file, which you can specify in Linker command file parameter. If you do not select this option, based on the device you have selected, a default custom linker command file will be used.

Linker command file

The path to memory description file that is required during linking. For each family of TI processor selected under 'Target Hardware', one linker command file will be selected automatically.

For different variant of processor, you can select from the 'src' folder inside the Support Package installation path. You can also create custom linker command file and select the file path using **Browse**.

CCS hardware configuration file

The Code Composer Studio file required for downloading the application on the hardware. Select one of the .ccxml files from the folder 'CCS_Config' folder under Support Package installation folder.

Instead, you can also create your own .ccxml file as explained in the section: “ ”. Select the file you created using **Browse**.

The .ccxml files provided with Embedded Coder Support Package for Texas Instruments C2000 Processors are as follows:

- f28027.ccxml— TI F28027 with Texas Instruments XDS100v1 USB Emulator
- f28035.ccxml— TI F28035 with Texas Instruments XDS100v1 USB Emulator
- f28069.ccxml— TI F28069 with Texas Instruments XDS100v1 USB Emulator
- f2808.ccxml—TI F2808 with Texas Instruments XDS100v1 USB Emulator
- f2808_eZdsp.ccxml—F2808 Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator
- f28044.ccxml—TI F28044 with Texas Instruments XDS100v1 USB Emulator
- f28335.ccxml—TI F28335 with Texas Instruments XDS100v1 USB Emulator
- f28335_eZdsp.ccxml—F28335 Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator
- f2812_BH2000.ccxml—Blackhawk USB2000 Controller for F2812 eZDSP
- f28x_generic.ccxml— Generic Texas Instruments XDS100v1 USB Emulator
- f28x_ezdsp_generic.ccxml—Generic Spectrum Digital eZdsp onboard USB Emulator
- f28055.ccxml— TI F28055 with Texas Instruments XDS100v1 USB Emulator

The .ccxml files provided with Embedded Coder Support Package for Texas Instruments C2000 F28M3x Concerto™ Processors are as follows:

- f28M35x.ccxml – Texas Instruments XDS100v2 USB Emulator_0
- f28M36x.ccxml – Texas Instruments XDS100v2 USB Emulator_0

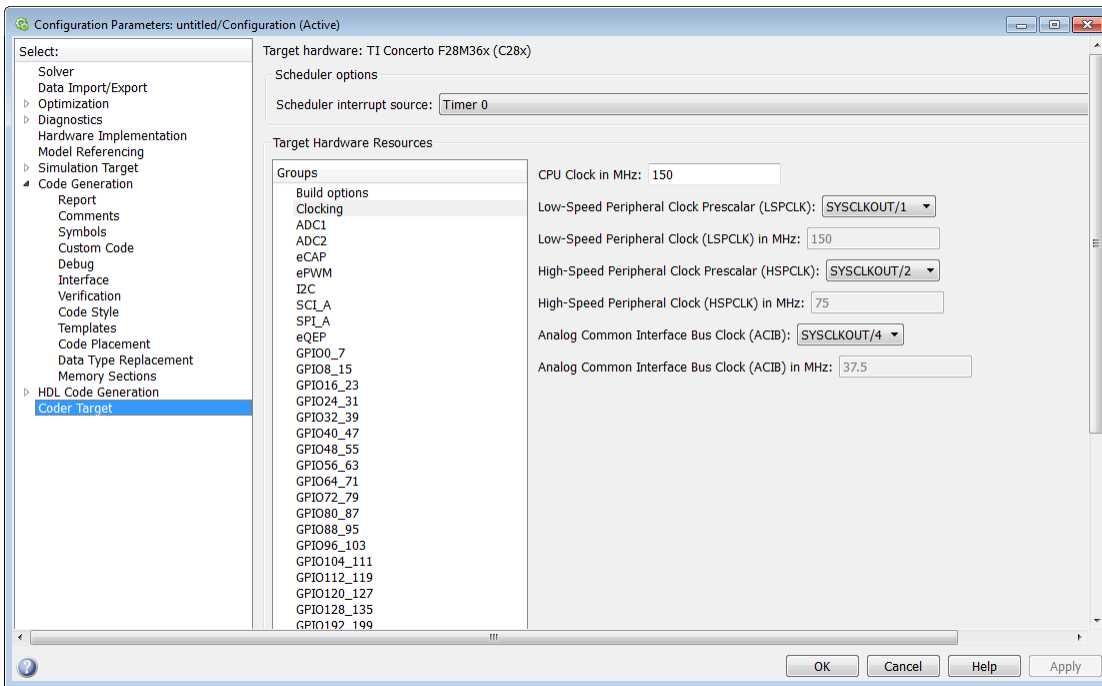
Enable DMA to access ePWM Registers instead of CLA

The option that you can select to enable the DMA to access ePWM registers instead of CLA. This option is available only for F2806x processors.

Remap ePWMs for DMA access (Requires silicon revision A and above)

The option that you can select to remap ePWMs registers for DMA access. This option is available only for F2833x processors.

Clocking



Use the clocking options to help you achieve the CPU Clock rate specified on the board. The default clocking values run the CPU clock (CLKIN) at its maximum frequency. The parameters use the external oscillator frequency on the board (OSCCLK) that is recommended by the processor vendor.

You can get feedback on the closest achievable SYSCLKOUT value with the specified Oscillator clock frequency by selecting the **Auto set PLL based on OSCCLK and CPU clock** check box. Alternatively, you can manually specify the PLL value for the SYSCLKOUT value calculation.

Change the clocking values if:

- You want to change the CPU frequency.
- The external oscillator frequency differs from the value recommended by the manufacturer.

To determine the CPU frequency (CLKIN), use the following equation:

$$\text{CLKIN} = (\text{OSCCLK} * \text{PLLCR}) / (\text{DIVSEL or CLKINDIV})$$

- CLKIN is the frequency at which the CPU operates, also known as the CPU clock.
- OSCCLK is the frequency of the oscillator.
- **PLLCR** is the PLL Control Register value.
- **CLKINDIV** is the Clock in Divider.
- **DIVSEL** is the Divider Select.

The availability of the DIVSEL or CLKINDIV parameters changes depending on the processor that you select. If neither parameter is available, use the following equation:

$$\text{CLKIN} = (\text{OSCCLK} * \text{PLLCR}) / 2$$

In the **CPU clock**

parameter of the Coder Target > Target Hardware Resources tab, enter the resulting CPU clock frequency (CLKIN).

For more information, see the “PLL-Based Clock Module” section in the Texas Instruments *Reference Guide* for your processor.

Use internal oscillator

Use the internal zero pin oscillator on the CPU. This parameter is enabled by default.

Oscillator clock (OSCCLK) frequency in MHz

The oscillator frequency that is used in the processor. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Auto set PLL based on OSCCLK and CPU clock

The option that helps you set the PLL control register value automatically. When you select this check box, the values in the PLLCR, DIVSEL, and the Closest achievable SYSCLKOUT in MHz parameters are automatically calculated based on the **CPU Clock** value entered on the Board. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

PLL control register (PLLCR)

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated control register value achieves the specified CPU Clock value, based on the Oscillator clock frequency. Otherwise, you can select a value for PLL control register. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Clock divider (DIVSEL)

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (DIVSEL). This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/DIVSEL
Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/CLKINDIV

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, PLLCR, and the DIVSEL. This parameter is not available for TI Concerto F28M35x/ F28M36x processors.

Low-Speed Peripheral Clock Prescaler (LSPCLK)

The value by which to scale the LSPCLK. This value is based on the SYSCLKOUT.

Low-Speed Peripheral Clock (LSPCLK) in MHz

This value is calculated based on LSPCLK Prescaler. Example: SPI uses a LSPCLK.

High-Speed Peripheral Clock Prescaler (HSPCLK)

The value by which to scale the HSPCLK. This value is based on the SYSCLKOUT.

High-Speed Peripheral Clock (HSPCLK) in MHz

This value is calculated based on HSPCLK Prescaler. Example: ADC uses a HSPCLK.

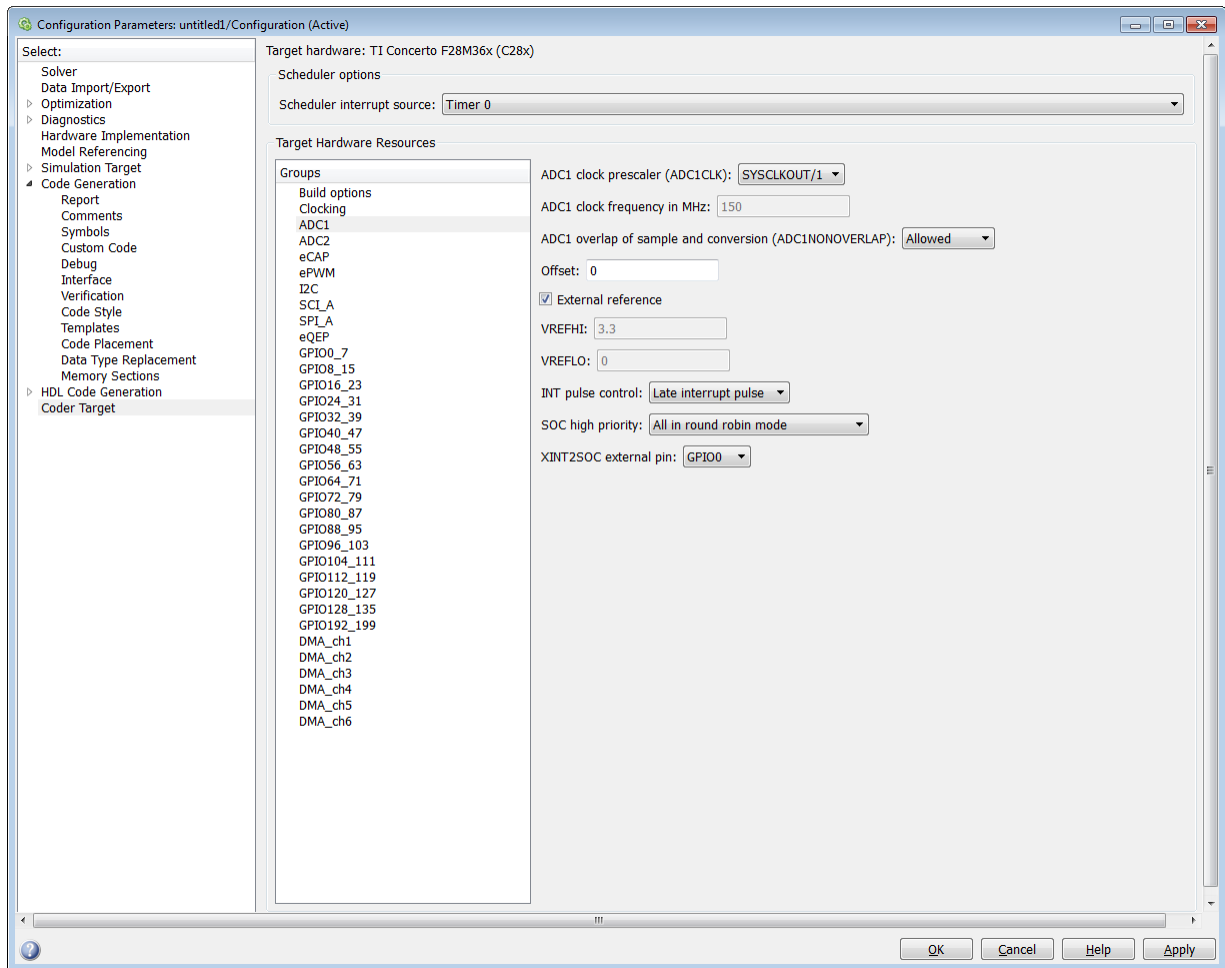
Analog Common Interface Bus Clock (ACIB)

The value by which to scale the bus clock. This option is available only for TI Concerto F28M35x/ F28M36x processors.

Analog Common Interface Bus Clock (ACIB) in MHz

This value is calculated based on the ACIB value. This option is available only for TI Concerto F28M35x/ F28M36x processors.

ADC



The high-speed peripheral clock (HSPCLK) controls the internal timing of the ADC module. The ADC derives the operating clock speed from the HSPCLK speed in several prescaler stages. For more information about configuring these scalars, refer to “Configuring ADC Parameters for Acquisition Window Width”.

You can set the following parameters for the ADC clock prescaler:

ADC clock prescaler (ADCCLK)

The option to select the ADCCLK divider for processors c2802x, c2803x, c2806x, or F28M3x.

ADC clock frequency in MHz

The clock frequency for ADC. This is a read-only field and the value in this field is based on the value you select in **ADC clock prescaler (ADCCLK)**.

ADC overlap of sample and conversion (ADC#NONOVERLAP)

The option to enable or disable overlap of sample and conversion.

ADC clock prescaler (ADCLKPS)

The HSPCLK speed is divided by this 4-bit value as the first step in deriving the core clock speed of the ADC. The default value is 3.

ADC Core clock prescaler (CPS)

After dividing the HSPCLK speed by the **ADC clock prescaler (ADCLKPS)** value, setting the **ADC clock prescaler (ADCLKPS)** parameter to 1, the default value, divides the result by 2.

ADC Module clock (ADCCLK = HSPCLK/ADCLKPS*2)/(CPS+1) in MHz

The clock to the ADC module and indicates the ADC operating clock speed.

Acquisition window prescaler (ACQ_PS)

This value does not directly alter the core clock speed of the ADC. It serves to determine the width of the sampling or acquisition period. The higher the value, the wider is the sampling period. The default value is 4.

Acquisition window size ((ACQ_PS+1)/ADCCLK) in micro seconds/channel

Acquisition window size determines for what time duration the sampling switch is closed. The width of SOC pulse is ADCTRL1[11:8] + 1 times the ADCLK period.

Offset

Enter the offset value.

Use external reference 2.048VExternal reference

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use a 2.048V external voltage reference.

Use external reference

By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable the external reference so the ADC logic uses an external voltage reference instead. Select the check box to use an external voltage reference.

Continuous mode

When the ADC generates an end of conversion (EOC) signal, generate an ADCINT# interrupt whether the previous interrupt flag has been acknowledged or not.

ADC offset correction (OFFSET_TRIM: -256 to 255)

The 280x ADC supports offset correction via a 9-bit value that it adds or subtracts before the results are available in the ADC result registers. Timing for results is not affected. The default value is 0.

VREFHI VREFLO

When you disable the **Use external reference 2.048V** or **External reference** option, the ADC logic uses a fixed 0-volt to 3.3-volt input range and the software disables **VREFHI** and **VREFLO**. To interpret the ADC input as a ratiometric signal, select the **External reference** option. Then set values for the high voltage reference (**VREFHI**) and the low voltage reference (**VREFLO**). **VREFHI** uses the external ADCINA0 pin, and **VREFLO** uses the internal GND.

INT pulse control

Use this option to configure when the ADC sets ADCINTFLG .ADCINTx relative to the SOC and EOC Pulses. Select **Late interrupt pulse** or **Early interrupt pulse**.

SOC high priority

Use this option to enable and configure **SOC high priority mode** . In all in round robin mode, the default selection, the ADC services each SOC interrupt in a numerical sequence.

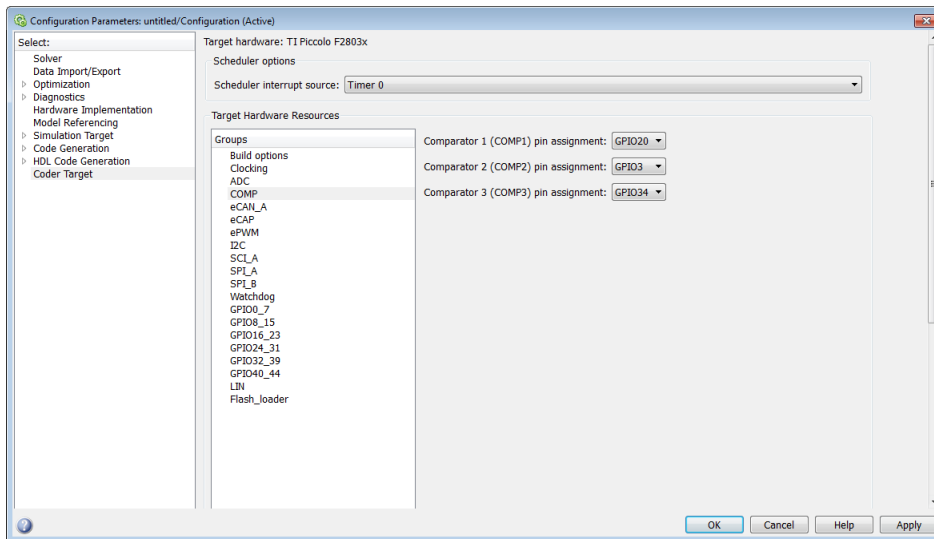
Choose one of the **high priority** selections to assign high priority to one or more of the SOC's. In this mode, the ADC operates in round robin mode until it receives a high priority SOC interrupt. The ADC finishes servicing the current SOC, services the high priority SOC's, and then returns to the next SOC in the round robin sequence.

For example, the ADC is servicing SOC8 when it receives a high priority interrupt on SOC1. The ADC completes servicing SOC8, services SOC1, and then services SOC9.

XINT2SOC external pin

Select the pin to which the ADC sends the XINT2SOC pulse.

COMP



Assigns COMP pins to GPIO pins.

Comparator 1 (COMP1) pin assignment

Select an option from the list — None,GPIO1, GPIO20, GPIO42.

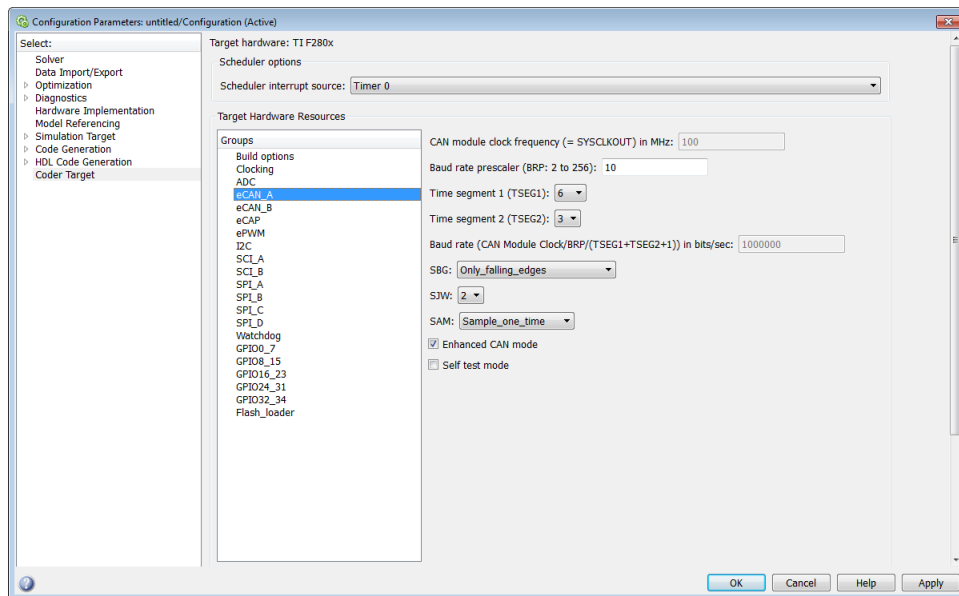
Comparator 2 (COMP2) pin assignment

Select an option from the list — None,GPIO3, GPIO21, GPIO34, GPIO43.

Comparator 3 (COMP3) pin assignment

Select an option from the list — None,GPIO34.

eCAN_A, eCAN_B



For more help on setting the timing parameters for the eCAN modules, refer to “Configuring Timing Parameters for CAN Blocks”. You can set the following parameters for the eCAN module:

CAN module clock frequency (= SYSCLKOUT) in MHz:

The clock to the enhanced CAN module. The CAN module clock frequency is equal SYSCLKOUT for processors such as c280x, c281x, c28044.

CAN module clock frequency (=SYSCLKOUT/2) in MHz

The clock to the enhanced CAN module. The CAN module clock frequency is equal to SYSCLKOUT/2 for processors such as piccolo, c2834x, c28x3x.

Baud rate prescaler (BRP: 2 to 256):

Value by which to scale the bit rate. Valid values are from 2 to 256.

Time segment 1 (TSEG1):

Sets the value of time segment 1, which, with **TSEG2** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG1** are from 1 through 16.

Time segment 2 (TSEG2):

Sets the value of time segment 2, which, with **TSEG1** and **Baud rate prescaler**, determines the length of a bit on the eCAN bus. Valid values for **TSEG2** are from 1 through 8.

Baud rate (CAN Module Clock/BRP/(TSEG1 + TSEG2 +1)) in bits/sec:

CAN module communication speed represented in bits/sec.

SBG

Sets the message resynchronization triggering. Options are `Only_falling_edges` and `Both_falling_and_rising_edges`.

SJW

Sets the synchronization jump width, which determines how many units of TQ a bit can be shortened or lengthened when resynchronizing.

SAM

Number of samples used by the CAN module to determine the CAN bus level. Selecting `Sample_one_time` samples once at the sampling point. Selecting `Sample_three_times` samples once at the sampling point and twice before at a distance of TQ/2. The CAN module makes a majority decision from the three points.

Enhanced CAN Mode

To enable time-stamping and to use **Mailbox Numbers** 16 through 31 in the C2000 eCAN blocks, enable this parameter. Texas Instruments documentation refers to this “HECC mode”.

Self test mode

If you set this parameter to `True`, the eCAN module goes to loopback mode. Loopback mode sends a “dummy” acknowledge message back without needing an acknowledge bit. The default is `False`.

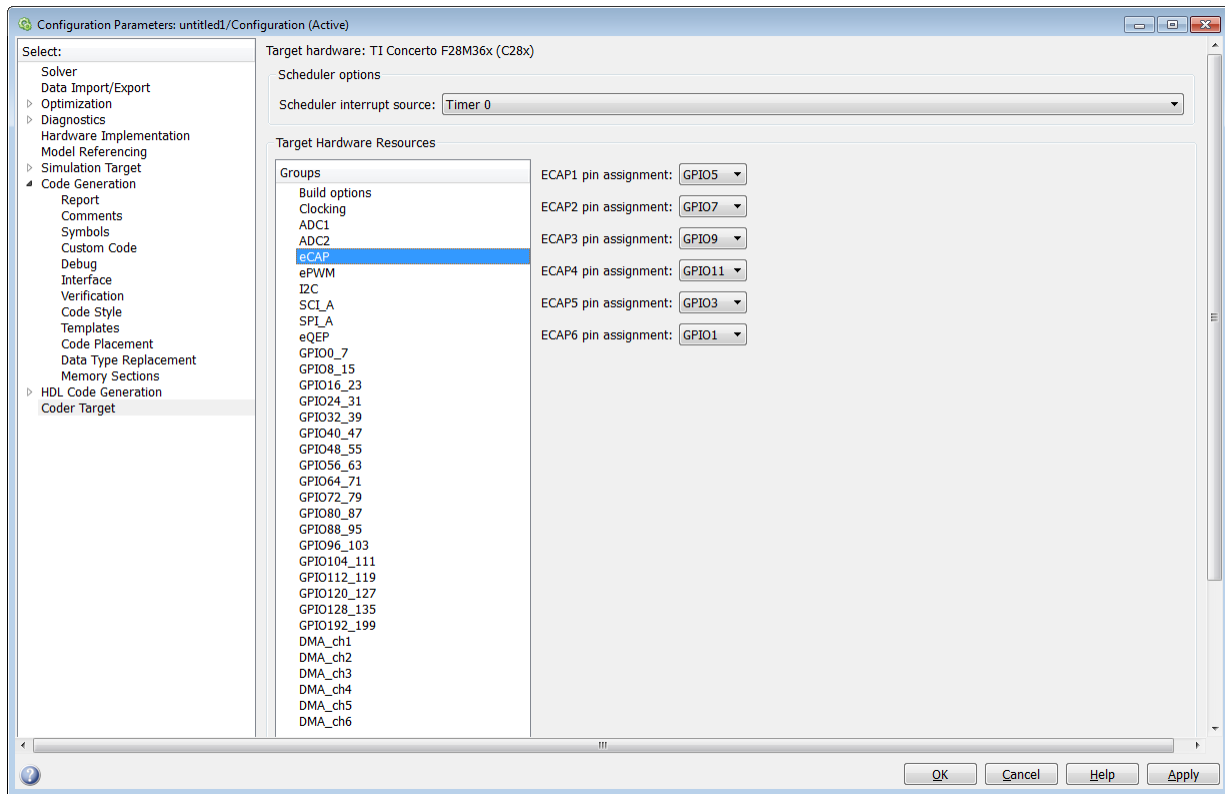
Pin assignment (Tx)

Assigns the CAN transmit pin to use with the eCAN_B module. Possible values are GPIO8, GPIO12, GPIO16, and GPIO20.

Pin assignment (Rx)

Assigns the CAN receive pin to use with the eCAN_B module. Possible values are GPIO10, GPIO13, GPIO17, and GPIO21.

eCAP



Assigns eCAP pins to GPIO pins.

ECAP1 pin assignment

Select an option from the list—None, GPIO5, or GPIO24 or GPIO34.

ECAP2 pin assignment

Select an option from the list—None, GPIO7, or GPIO25 or GPIO37.

ECAP3 pin assignment

Select an option from the list—None, GPIO9, or GPIO26.

ECAP4 pin assignment

Select an option from the list—None, GPIO11, or GPIO27.

ECAP5 pin assignment

Select an option from the list—None, GPIO3, or GPIO48.

TZ3 pin assignment

Assigns the trip-zone input 3 (TZ3) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO29.

TZ4 pin assignment

Assigns the trip-zone input 4 (TZ4) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO28.

TZ5 pin assignment

Assigns the trip-zone input 5 (TZ5) to a GPIO pin. Choices are None (the default), GPIO16, and GPIO28.

TZ6 pin assignment

Assigns the trip-zone input 6 (TZ6) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO29.

Note The TZ# pin assignments are available only for TI F280x processors.

SYNCI pin assignment

Assigns the ePWM external sync pulse input (SYNCI) to a GPIO pin. Choices are None (the default), GPIO6, and GPIO32.

SYNCO pin assignment

Note SYNCI and SYNCO pin assignments are available for TI F28044, TI F280x, TI Delfino F2833x, TI Delfino F2834x, TI Piccolo F2802x, TI Piccolo F2803x, TI Piccolo F2806 processors.

Assigns the ePWM external sync pulse output (SYNCO) to a GPIO pin. Choices are None (the default), GPIO6, and GPIO33.

PWM#A, PWM#B, PWM#C pin assignment

The PWM # A, PWM # B, PWM # C pin assignment.

GPTRIP#SEL pin assignment

Assigns the ePWM trip-zone input to a GPIO pin.

Note The GPTRIP#SEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

PWM1SYNCl/ GPTRIP6SEL pin assignment

Assigns the ePWM sync pulse input (SYNCl) to a GPIO pin.

Note: The PWM1SYNCl/GPTRIP#SEL pin assignments are available only for TI Concerto F28M35x/F28M36x processors.

DCAHTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBHTRIPSEL (Enter Hex value between 0 and 0x6FFF)

Assigns the Digital Compare A High Trip Input to a GPIO pin.

Note DCAHTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

DCALTRIPSEL (Enter Hex value between 0 and 0x6FFF) / DCBLTRIPSEL (Enter Hex value between 0 and 0x6FFF)

Assigns the Digital Compare A High Trip Input to a GPIO pin.

Note The DCALTRIPSEL pin assignment is available only for TI Concerto F28M35x/F28M36x processors.

I2C

If a module is an I2C slave, it:

- Synchronizes itself with the serial clock line (SCL) line.
- Responds to communication requests from the master.

When **Mode** is **Slave**, you can configure the **Addressing format**, **Address register**, and **Bit count** parameters.

The **Mode** parameter corresponds to bit 10 (MST) of the I2C Mode Register (I2CMDR).

Addressing format

If **Mode** is **Slave**, determine the addressing format of the I2C master, and set the I2C module to the same mode:

- **7-Bit Addressing**, the normal address mode.
- **10-Bit Addressing**, the expanded address mode.
- **Free Data Format**, a mode that does not use addresses. (If you **Enable loopback**, the Free data format is not supported.)

The **Addressing format** parameter corresponds to bit 3 (FDF) and bit 8 (XA) of the I2C Mode Register (I2CMDR).

Own address register

If **Mode** is **Slave**, enter the 7-bit (0–127) or 10-bit (0–1023) address this I2C module uses while it is a slave.

This parameter corresponds to bits 9–0 (OAR) of the I2C Own Address Register (I2COAR).

Bit count

If **Mode** is **Slave**, set the number of bits in each data *byte* the I2C module transmits and receives. This value must match that of the I2C master.

This parameter corresponds to bits 2–0 (BC) of the I2C Mode Register (I2CMDR).

Module clock prescaler (IPSC: 0 to 255):

If **Mode** is **Master**, configure the module clock frequency by entering a value 0–255.

Module clock frequency = I2C input clock frequency / (Module clock prescaler + 1)

The I2C specifications require a module clock frequency between 7 MHz and 12 MHz.

The *I2C input clock frequency* depends on the DSP input clock frequency and the value of the PLL Control Register divider (PLLCR). For more information on setting the PLLCR, consult the documentation for your specific Digital Signal Controller.

This **Module clock prescaler (IPSC: 0 to 255)**: corresponds to bits 7–0 (IPSC) of the I2C Prescaler Register (I2CPSC).

I2C Module clock frequency (SYSCLKOUT / (IPSC+1)) in Hz:

This field displays the frequency the I2C module uses internally. To set this value, change the **Module clock prescaler**.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721, on the Texas Instruments Web site.

I2C Master clock frequency (Module Clock Freq/(ICCL+ICCH+10)) in Hz:

This field displays the master clock frequency.

For more information about this value, consult the “Clock Generation” section in the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

Master clock Low-time divider (ICCL: 1 to 65535):

When **Mode** is **Master**, this divider determines the duration of the low state of the SCL line on the I2C-bus.

The low-time duration of the master clock = $T_{\text{mod}} \times (\text{ICCL} + d)$.

For more information, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A/ SPRUH22F/ SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCL) of the Clock Low-Time Divider Register (I2CCLKL).

Master clock High-time divider (ICCH: 1 to 65535):

When **Mode** is **Master**, this divider determines the duration of the high state on the serial clock pin (SCL) of the I2C-bus.

The high-time duration of the master clock = $T_{\text{mod}} \times (\text{ICCL} + d)$.

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x/ TMS320F28M3x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721A, SPRUH22f, SPRUHE8B, available on the Texas Instruments Web site.

This parameter corresponds to bits 15–0 (ICCH) of the Clock High-Time Divider Register (I2CCLKH).

Enable loopback

When **Mode** is **Master**, enable or disable digital loopback mode. In digital loopback mode, I2CDXR transmits data over an internal path to I2CDRR, which receives the data after a configurable delay.

The delay, measured in DSP cycles, equals (I2C input clock frequency/module clock frequency) x 8.

While **Enable loopback** is enabled, free data format addressing is not supported.

This parameter corresponds to bit 6 (DLB) of the I2C Mode Register (I2CMODR).

SDAA pin assignment

Select the GPIO pin to use on the SDAA boot function.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

SCLA pin assignment

Select the GPIO pin to use on the SCLA boot function.

This parameter is available only for TI Concerto F28M36x (C28x) processors.

Enable Tx interrupt

This parameter corresponds to bit 5 (TXFFIENA) of the I2C Transmit FIFO Register (I2CFFTX).

Tx FIFO interrupt level

This parameter corresponds to bits 4–0 (TXFFIL4-0) of the I2C Transmit FIFO Register (I2CFFTX).

Enable Rx interrupt

This parameter corresponds to bit 5 (RXFFIENA) of the I2C Receive FIFO Register (I2CFFRX).

Rx FIFO interrupt level

This parameter corresponds to bit 4–0 (RXFFIL4-0) of the I2C Receive FIFO Register (I2CFFRX).

Enable system interrupt

Select this parameter to display and individually configure the following five Basic I2C Interrupt Request parameters in the Interrupt Enable Register (I2CIER):

- Enable AAS interrupt
- Enable SCD interrupt
- Enable ARDY interrupt
- Enable NACK interrupt
- Enable AL interrupt

Enable AAS interrupt

Enable the addressed-as-slave interrupt.

When enabled, the I2C module generates an interrupt (AAS bit = 1) upon receiving one of the following:

- Its **Own address register**
- A general call (all zeros)
- A data byte is in free data format

When enabled, the I2C module clears the interrupt (AAS = 0) upon receiving one of the following:

- Multiple START conditions (7-bit addressing mode only)
- A slave address that is different from **Own address register** (10-bit addressing mode only)
- A NACK or a STOP condition

This parameter corresponds to bit 6 (AAS) of the Interrupt Enable Register (I2CIER).

Enable SCD interrupt

Enable stop condition detected interrupt.

When enabled, the I2C module generates an interrupt (SCD bit = 1) when the CPU detects a stop condition on the I2C bus.

When enabled, the I2C module clears the interrupt (SCD = 0) upon one of the following events:

- The CPU reads the I2CISRC while it indicates a stop condition
- A reset of the I2C module
- Someone manually clears the interrupt

This parameter corresponds to bit 5 (SCD) of the Interrupt Enable Register (I2CIER).

Enable ARDY interrupt

Enable register-access-ready interrupt enable bit.

When enabled, the I2C module generates an interrupt (ARDY bit = 1) when the previous address, data, and command values in the I2C module registers have been used and new values can be written to the I2C module registers.

This parameter corresponds to bit 2 (ARDY) of the Interrupt Enable Register (I2CIER).

Enable NACK interrupt

Enable no acknowledgment interrupt enable bit.

When enabled, the I2C module generates an interrupt (NACK bit = 1) when the module is a transmitter in master or slave mode and it receives a NACK condition.

This parameter corresponds to bit 1 (NACK) of the Interrupt Enable Register (I2CIER).

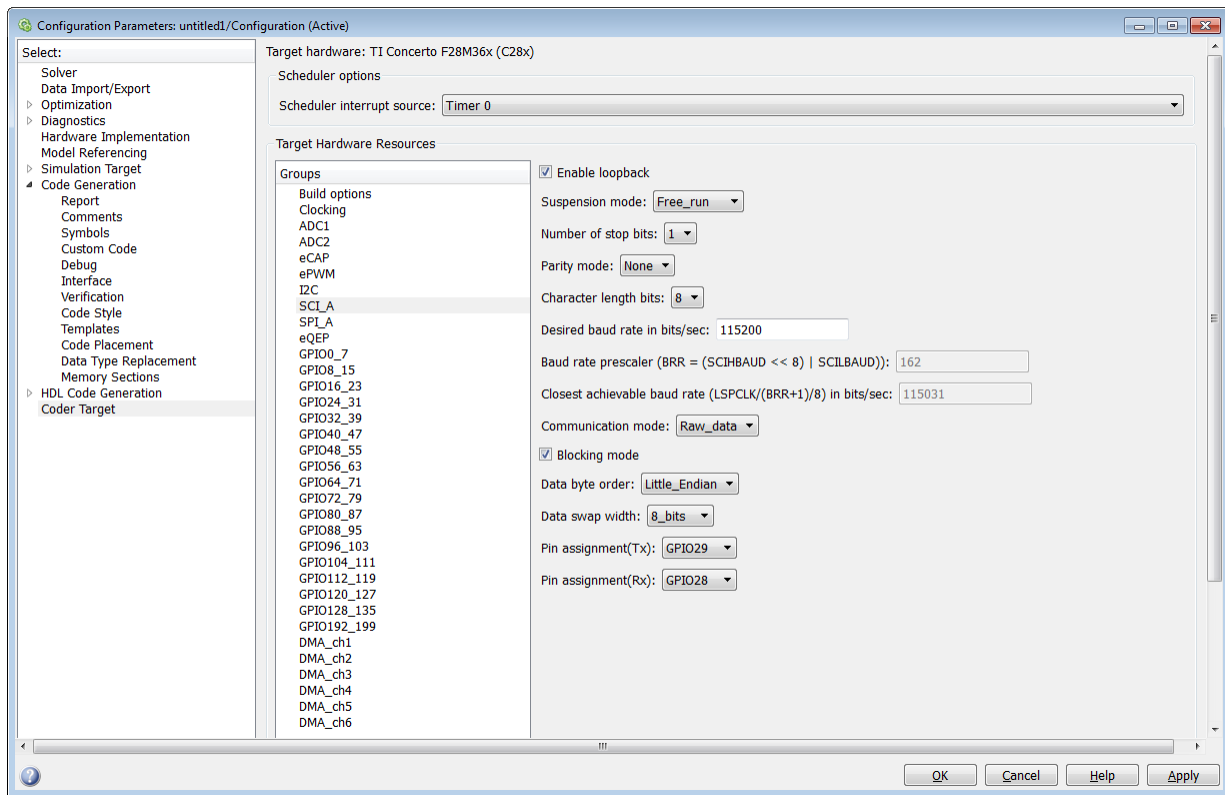
Enable AL interrupt

Enable arbitration-lost interrupt.

When enabled, the I2C module generates an interrupt (AL bit = 1) when the I2C module is operating as a master transmitter and loses an arbitration contest with another master transmitter.

This parameter corresponds to bit 0 (AL) of the Interrupt Enable Register (I2CIER).

SCI_A, SCI_B, SCI_C



The serial communications interface parameters you can set for module A. These parameters are:

Enable loopback

Select this parameter to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, a C28x DSP Tx pin is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

Baud rate

Baud rate for transmitting and receiving data. Select from 115200 (the default), 57600, 38400, 19200, 9600, 4800, 2400, 1200, 300, and 110.

Suspension mode

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive/transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

Number of stop bits

Select whether to use 1 or 2 stop bits.

Parity mode

Type of parity to use. Available selections are `None`, `Odd parity`, or `Even parity`. `None` disables parity. `Odd` sets the parity bit to one if you have an odd number of ones in your bytes, such as 00110010. `Even` sets the parity bit to one if you have an even number of ones in your bytes, such as 00110011.

Character length bits

Length in bits of each transmitted or received character, set to 8 bits.

Desired baud rate in bits/sec

The desired baud rate specified by the user.

Baud rate prescaler (BRR = (SCIHBAUD << 8) | SCILBAUD)

The baud rate prescaler.

Closest achievable baud rate (LSPCLK/(BRR+1)/8) in bits/sec

The closest achievable baud rate calculated based on LSPCLK and BRR.

Communication mode

Select `Raw_data` or `Protocol` mode. Raw data is unformatted and sent whenever the transmitting side is ready to send, whether the receiving side is ready or not. Without a wait state, deadlock conditions do not occur. Data transmission is asynchronous. With this mode, it is possible the receiving side could miss data, but if the data is noncritical, using raw data mode can avoid blocking processes.

When you select protocol mode, some handshaking between host and processor occurs. The transmitting side sends `$SND` to indicate it is ready to transmit. The receiving side sends back `$RDY` to indicate it is ready to receive. The transmitting side then sends data and, when the transmission is completed, it sends a checksum.

Advantages to using protocol mode include:

- Avoids deadlock

- Determines whether data is received without errors (checksum)
- Determines whether data is received by processor
- Determines time consistency; each side waits for its turn to send or receive

Note Deadlocks can occur if one SCI Transmit block tries to communicate with more than one SCI Receive block on different COM ports when both are blocking (using protocol mode). Deadlocks cannot occur on the same COM port.

Blocking mode

If this option is set to `True`, system waits until data is available to read (when data length is reached). If this option is set to `False`, system checks FIFO periodically (in polling mode) for data to read. If data is present, the block reads and outputs the contents. When data is not present, the block outputs the last value and continues.

Data byte order

Select `Little Endian` or `Big Endian`, to match the endianness of the data being moved.

Data swap width

Select `8_bits` or `16_bits`, to match the width of the data being moved by the data swap operation. When you set **Data byte order** to `Big Endian`, the only available option for **Data swap width** is `8_bits`.

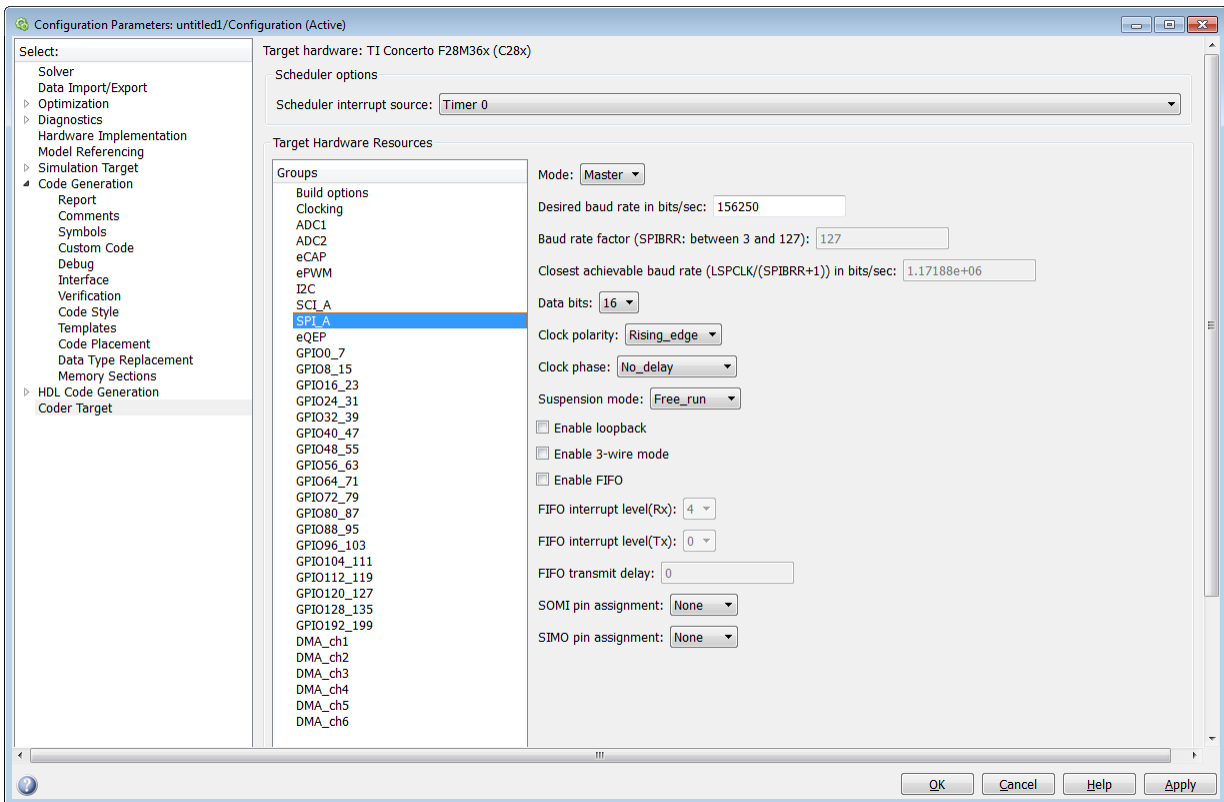
Pin assignment (Tx)

Assigns the SCI transmit pin to use with the SCI module.

Pin assignment (Rx)

Assigns the SCI receive pin to use with the SCI module.

SPI_A, SPI_B, SPI_C, SPI_D



The serial peripheral interface parameters you can set for the A module. These parameters are:

Mode

Set to **Master** or **Slave**.

Desired baud rate in bits/sec

The desired baud rate specified by the user.

Baud rate factor (SPIBRR: between 3 and 127)

To set the **Baud rate factor**, search for “Baud Rate Determination” and “SPI Baud Rate Register (SPIBRR) Bit Descriptions” in *TMS320x28xx, 28xxx DSP Serial Peripheral Interface (SPI) Reference Guide*, Literature Number: SPRU059, available on the Texas Instruments Web Site.

Closest achievable baud rate (LSPCLK/(SPIBRR+1)) in bits/sec

The closest achievable baud rate calculated based on LSPCLK and SPIBRR.

Data bits

Length in bits from 1 to 16 of each transmitted or received character. For example, if you select 8, the maximum data that can be transmitted using SPI is 2^{8-1} . If you send data greater than this value, the buffer overflows.

Clock polarity

Select `Rising_edge` or `Falling_edge`.

Clock phase

Select `No_delay` or `Delay_half_cycle`.

Suspension mode

Type of suspension to use when debugging your program with Code Composer Studio. When your program encounters a breakpoint, the selected suspension mode determines whether to perform the program instruction. Available options are `Hard_abort`, `Soft_abort`, and `Free_run`. `Hard_abort` stops the program immediately. `Soft_abort` stops when the current receive or transmit sequence is complete. `Free_run` continues running regardless of the breakpoint.

Enable loopback

Select this option to enable the loopback function for self-test and diagnostic purposes only. When this function is enabled, the Tx pin on a C28x DSP is internally connected to its Rx pin and can transmit data from its output port to its input port to check the integrity of the transmission.

Enable 3-wire mode

Enable SPI communication over three pins instead of the normal four pins.

Enable FIFO

Set true or false.

FIFO interrupt level (Rx)

Set level for receive FIFO interrupt. Select 0 through 16.

FIFO interrupt level (Tx)

Set level for transmit FIFO interrupt. Select 0 through 16.

FIFO transmit delay

Enter FIFO transmit delay (in processor clock cycles) to pause between data transmissions. Enter an integer.

CLK pin assignment

Assigns the SPI something (CLK) to a GPIO pin. Choices are **None** (default), **GPI014**, or **GPI026**.

CLK pin assignment is not available for TI Concerto F28M35x/F28M36x processors.

SOMI pin assignment

Assigns the SPI something (SOMI) to a GPIO pin. Choices are **None** (default), **GPI013**, or **GPI025**.

STE pin assignment

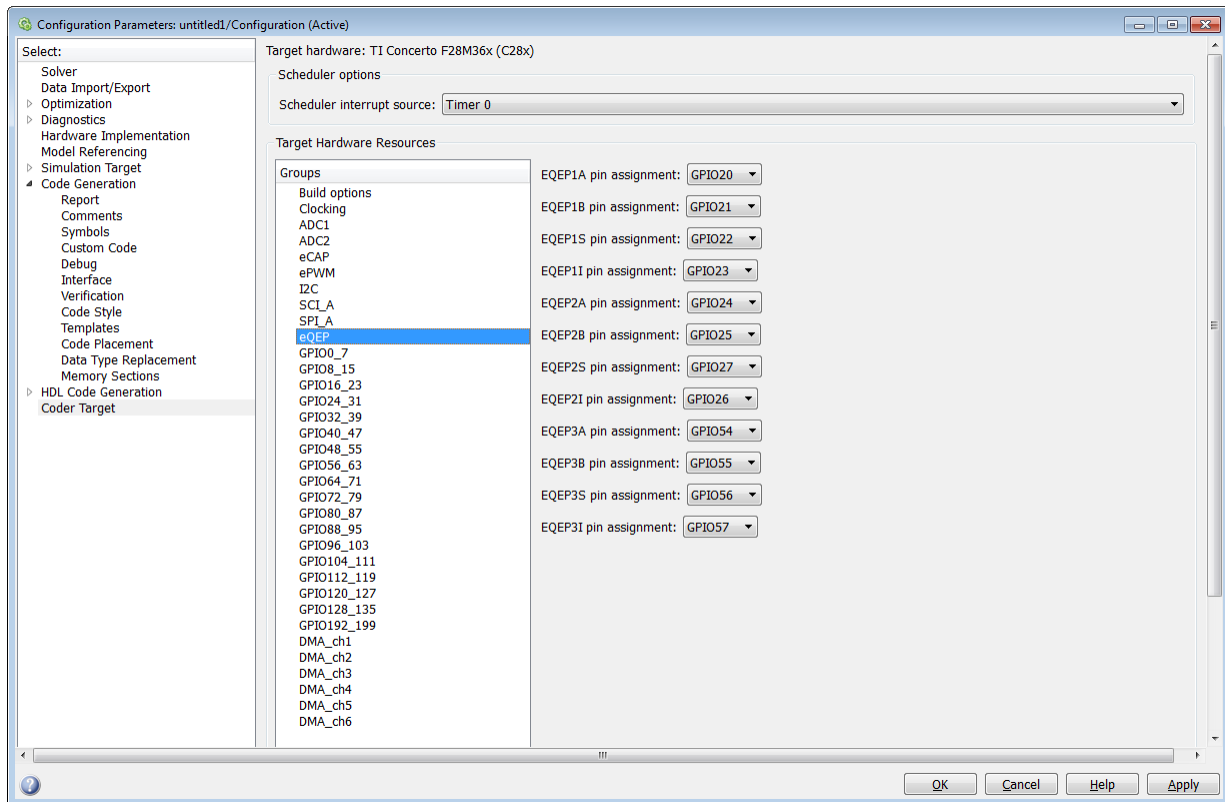
Assigns the SPI something (STE) to a GPIO pin. Choices are **None** (default), **GPI015**, or **GPI027**.

STE pin assignment is not available for TI Concerto F28M35x/ F28M36x processors.

SIMO pin assignment

Assigns the SPI something (SIMO) to a GPIO pin. Choices are **None** (default), **GPI012**, or **GPI024**.

eQEP



Assigns eQEP pins to GPIO pins.

EQEP1A pin assignment

Select an option from the list—None, GPIO20, GPIO50, GPIO64, GPIO106.

EQEP1B pin assignment

Select an option from the list—None, GPIO21, GPIO51, GPIO65, GPIO107.

EQEP1S pin assignment

Select an option from the list—None, GPIO22, GPIO52, GPIO66, GPIO108.

EQEP1I pin assignment

Select an option from the list—None, GPIO23, GPIO53, GPIO67, GPIO109.

EQEP2A pin assignment

Select an option from the list—None, GPIO24, GPIO54, GPIO66, GPIO110. The pin numbers shown in the list vary based on the processor selected.

EQEP2B pin assignment

Select an option from the list—None, GPIO25, GPIO55, GPIO67, GPIO111. The pin numbers shown in the list vary based on the processors selected.

EQEP2S pin assignment

Select an option from the list—None, GPIO27, GPIO31, GPIO57, GPIO67, GPIO113.

EQEP2I pin assignment

Select an option from the list—None, GPIO26, GPIO30, GPIO56, GPIO64, GPIO112.

EQEP3A pin assignment

Select an option from the list—None or GPIO54, or GPIO112. This parameter is available only for TI Concerto F28M36x (C28x) processors.

EQEP3B pin assignment

Select an option from the list—None or GPIO55, or GPIO113. This parameter is available only for TI Concerto F28M36x (C28x) processors.

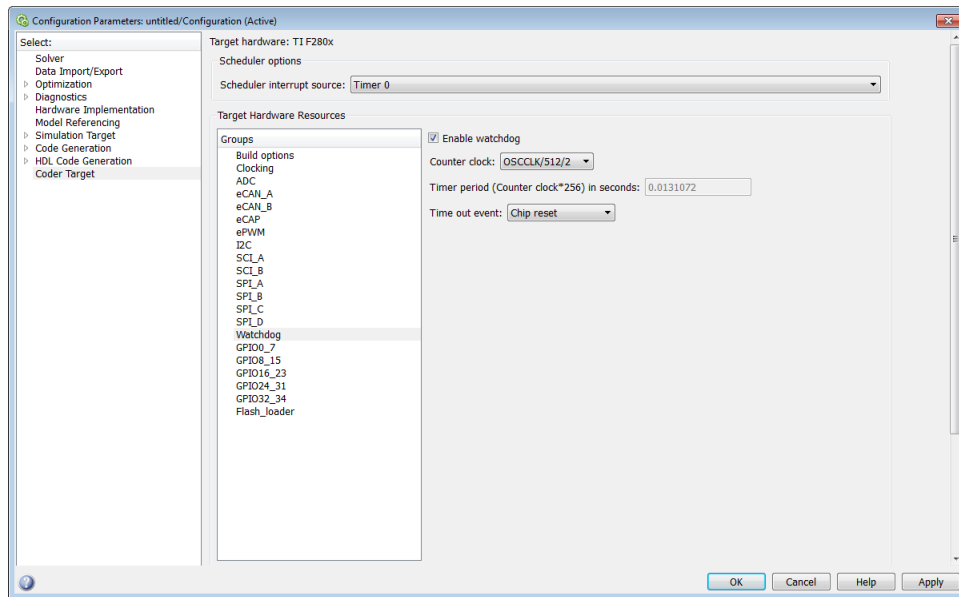
EQEP3S pin assignment

Select an option from the list—None, or GPIO56, or GPIO110

EQEP3I pin assignment

Select an option from the list—None, or GPIO57, or GPIO111

Watchdog



When enabled, if the software fails to reset the watchdog counter within a specified interval, the watchdog resets the processor or generates an interrupt. This feature enables the processor to recover from some fault conditions.

For more information, locate the *Data Manual* or *System Control and Interrupts Reference Guide* for your processor on the Texas Instruments Web site.

Enable watchdog

Enable the watchdog timer module.

This parameter corresponds to bit 6 (WDDIS) of the Watchdog Control Register (WDCR) and bit 0 (WDOVERRIDE) of the System Control and Status Register (SCSR).

Counter clock

Set the watchdog timer period relative to OSCCLK/512.

This parameter corresponds to bits 2–0 (WDPS) of the Watchdog Control Register (WDCR).

Timer period (Counter clock*256) in seconds

This field displays the timer period in seconds. This value automatically updates when you change the **Counter clock** parameter.

Time out event

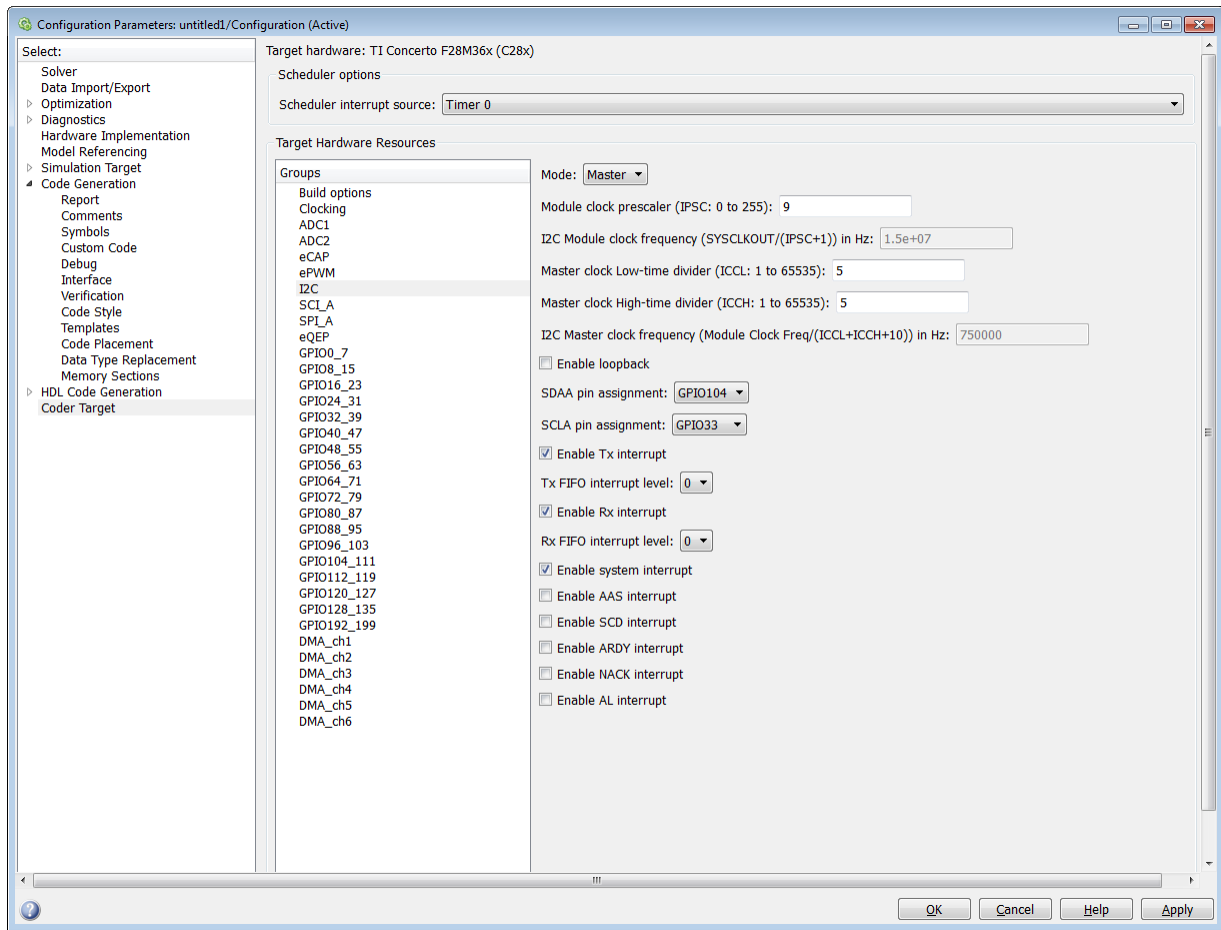
Configure the watchdog to reset the processor or generate an interrupt when the software fails to reset the watchdog counter:

- Select **Chip reset** to generate a signal that resets the processor (WDRST signal) and disable the watchdog interrupt signal (WDINT signal).
- Select **Raise WD Interrupt** to generate a watchdog interrupt signal (WDINT signal) and disable the reset processor signal (WDRST signal). This signal can be used to wake the device from an IDLE or STANDBY low-power mode.

This parameter corresponds to bit 1 (WDENINT) of the System Control and Status Register (SCSR).

GPIO

3 Configuration Parameters



GPIO Use the GPIO pins for digital input or output by connecting to one of the three peripheral I/O ports.

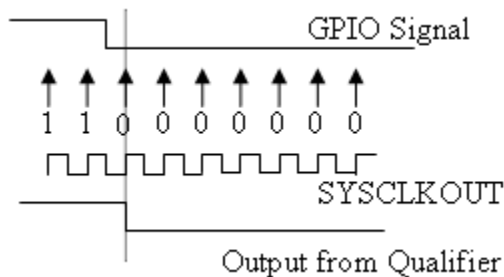
The range of GPIO pins for different processors is given below:

Processors	GPIO Pin Values
C281x	GPIOA, GPIOB, GPIOD, GPIOE, GPIOF, and GPIOG
F2803x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44

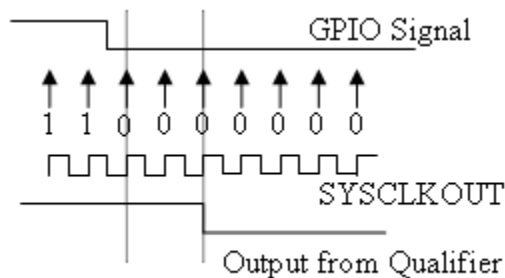
Processors	GPIO Pin Values
F2806x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–44, GPIO50–55, GPIO56–58
F2823x, F2833x, and C2834x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48–55, GPIO56–63
C2801x, F2802x, F28044, F280x	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–34
F28M35x (C28x)	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO32–39, GPIO40–47, GPIO48_55, GPIO56–63, GPIO68–71, GPIO128–135
F28M36x (C28x)	GPIO0–7, GPIO8–15, GPIO16–23, GPIO24–31, GPIO40–47, GPIO48–55, GPIO56–63, GPIO64–71, GPIO72–79, GPIO80–87, GPIO88–95, GPIO96–103, GPIO104–111, GPIO112–119, GPIO120–127, GPIO128–135, GPIO192–199.

Each pin selected for input offers four signal qualification types:

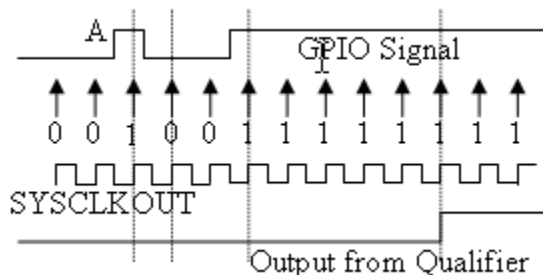
- **Sync to SYSCLKOUT only** — This setting is the default for all pins at reset. Using this qualification type, the input signal is synchronized to the system clock SYSCLKOUT. The following figure shows the input signal measured on each tick of the system clock, and the resulting output from the qualifier.



- **Qualification using 3 samples** — This setting requires three consecutive cycles of the same value for the output value to change. The following figure shows that, in the third cycle, the GPIO value changes to 0, but the qualifier output is still 1 because it waits for three consecutive cycles of the same GPIO value. The next three cycles all have a value of 0, and the output from the qualifier changes to 0 immediately after the third consecutive value is received.



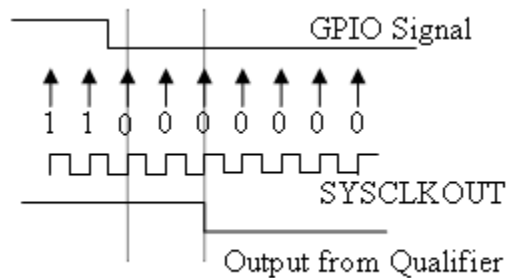
- Qualification using 6 samples** — This setting requires six consecutive cycles of the same GPIO input value for the output from the qualifier to change. In the following figure, the glitch **A** does not alter the output signal. When the glitch occurs, the counting begins, but the next measurement is low again, so the count is ignored. The output signal does not change until six consecutive samples of the high signal are measured.



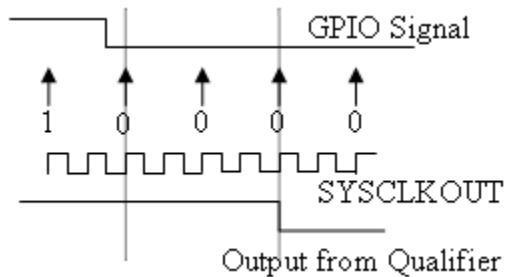
Qualification sampling period prescaler

Visible only when a setting for **Qualification type for GPIO [pin#]** is selected. The qualification sampling period prescaler, with possible values of 0 to 255, calculates the frequency of the qualification samples or the number of system clock ticks per sample. The formula for calculating the qualification sampling frequency is $\text{SYSCLKOUT}/(2 * \text{Prescaler})$, except for zero. When **Qualification sampling period prescaler=0**, a sample is taken every SYSCLKOUT clock tick. For example, a prescale setting of 0 means that a sample is taken on each SYSCLKOUT tick.

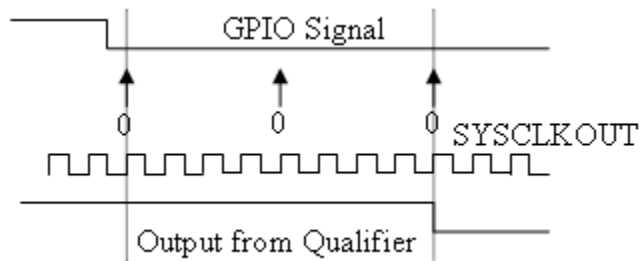
The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to **Qualification** using 3 samples. In this case, the **Qualification sampling period prescaler=0**:



In the next figure **Qualification sampling period prescaler=1**. A sample is taken every two clock ticks, and the **Qualification type** is set to **Qualification** using 3 samples. The output signal changes much later than if **Qualification sampling period prescaler=0**.



In the following figure, **Qualification sampling period prescaler=2**. Thus , a sample is taken every four clock ticks, and the **Qualification type** is set to **Qualification** using 3 samples.



- Asynchronous

Using this qualification type, the signal is synchronized to an asynchronous event initiated by software (CPU) via control register bits.

Qualification sampling period

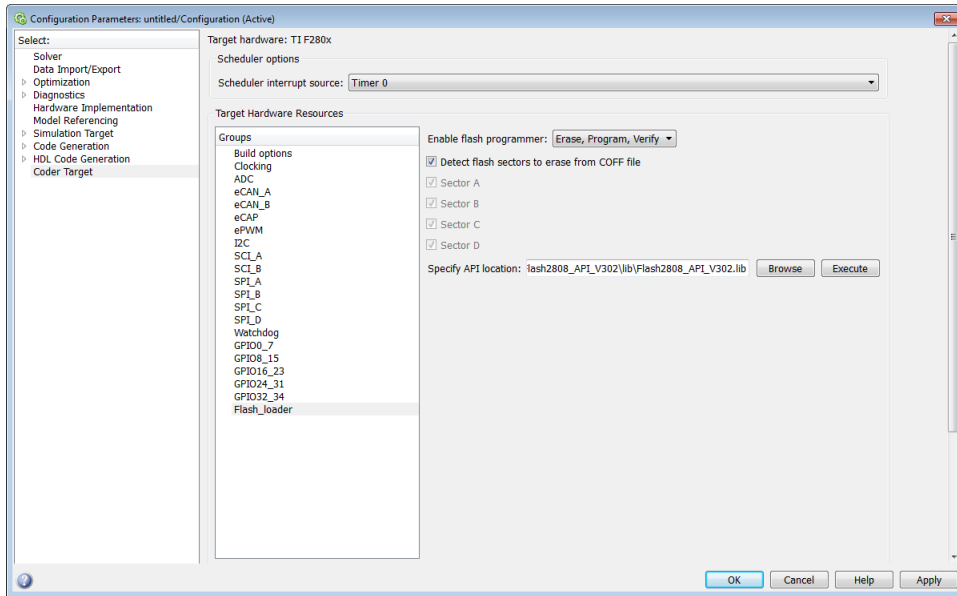
Enter the qualification sampling period.

GPIOA, GPIOB, GPIOD, GPIOE input qualification sampling period

GPIO# Pull Up Disabled

Select this check box to disable the GPIO pull up register. This option is available only for TI Concerto F28M35x/F28M36x processors.

Flash_loader



You can use `Flash_loader` to:

- Automatically erase program generated code to flash memory on the target when you build the code.
- Manually erase, program, or verify specific flash memory sectors.

To use this feature, download and install the TI Flash API plugin from the TI Web site.

For more information, consult the “Programming Flash Memory” topic or the `*_API_Readme.pdf` file included in the *TI Flash API* downloadable zip file.

Enable Flash Programmer

Enable the flash programmer by selecting a task for it to perform when you click **Execute** or build the software.

To program the flash memory when you build the software, select **Erase**, **Program**, **Verify**.

Detect Flash sectors to erase from COFF file

When enabled, the flash programmer erases all of the flash sectors defined by the COFF file.

Sector A, Sector B, Sector C...

When **Detect Flash sectors to erase from COFF file** is disabled, you can select the specific sector to erase.

Specify API location

Specify the folder path of the TI flash API executable you downloaded and installed on your computer.

Use **Browse** to locate the file or enter the path in the text box.

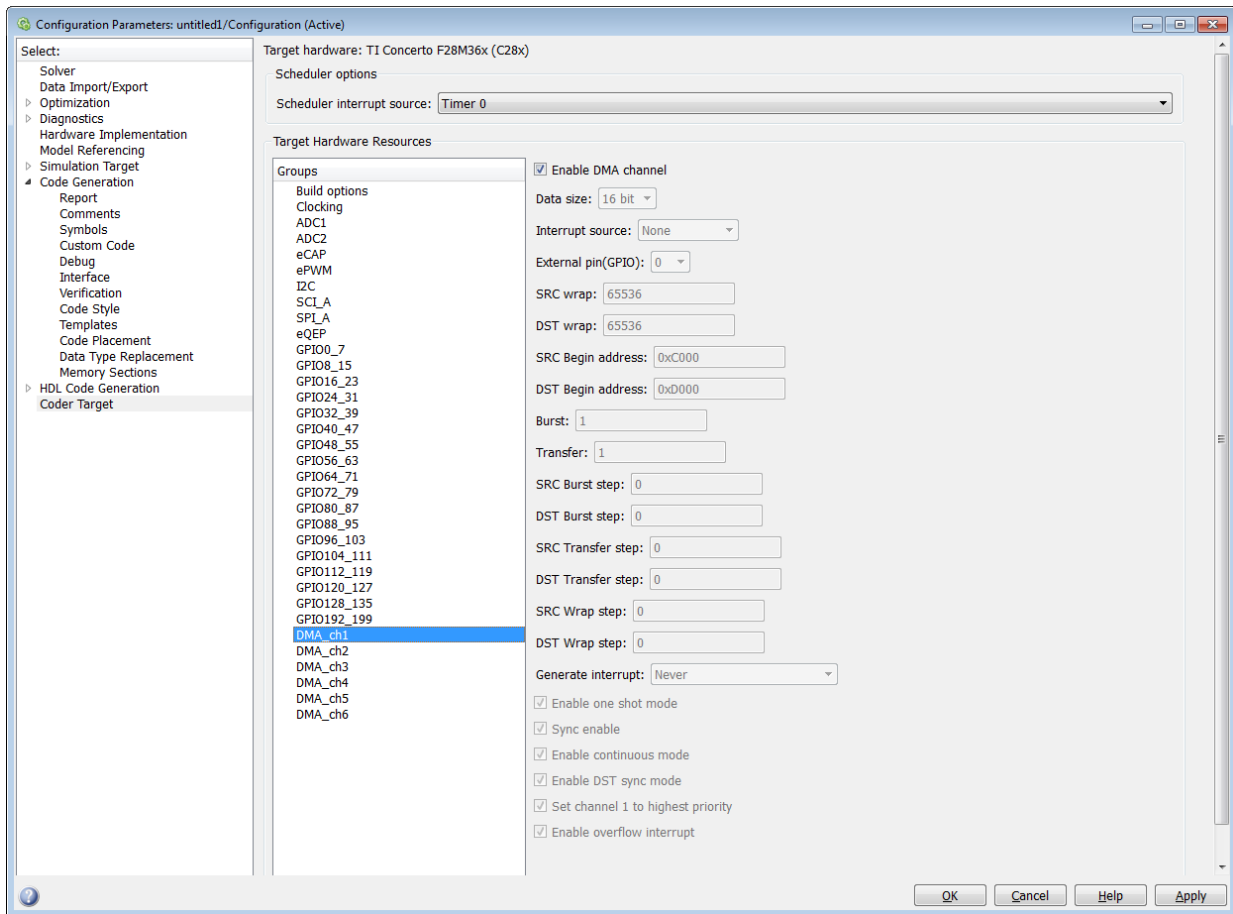
For example,

```
C:\TI\controlSUITE\libs\utilities\flash_api\2806x\v100\lib\2806x_BootROM_API_TABLE_Symbols_fpu32.lib
```

Execute

Click this button to initiate the task selected in **Enable Flash Programmer**.

DMA_ch[#]



The Direct Memory Access module transfers data directly between peripherals and memory using a dedicated bus, increasing overall system performance.

You can individually enable and configure each DMA channel.

The DMA module services are event driven. Using the Interrupt source and **External pin (GPIO)** parameters, you can configure a wide range of peripheral interrupt event triggers.

To avoid error messages, open the Coder Target > Target Hardware Resources, select the **Peripherals** tab, and *disable* the same DMA channel number.

For more information, consult the *TMS320x2833x, 2823x/ TMS320F28M3x Direct Memory Access (DMA) Module Reference Guide*, Literature Number: SPRUFB8A/SPRUH22F/ SPRUHE8B.

Also consult the *Increasing Data Throughput using the TMS320F2833x DSC DMA* training presentation (requires login), both available from the TI Web site.

Enable DMA channel

Enable this parameter to edit the configuration of a specific DMA channel.

This parameter does not have a corresponding bit or register.

Data size

Select the size of the data bit transfer: **16 bit** or **32 bit**.

The DMA read/write data buses are 32 bits wide. 32-bit transfers have twice the data throughput of a 16-bit transfer.

When providing DMA service to McBSP, set **Data size** to **16 bit**.

The following parameters are based on a 16-bit word size. If you set **Data size** to **32 bit**, double the value of the following parameters:

- Size: Burst
- Source: Burst step
- Source: Transfer step
- Source: Wrap step
- Destination: Burst step
- Destination: Transfer step
- Destination: Wrap step

Data size corresponds to bit 14 (DATASIZE) in the Mode Register (MODE).

Interrupt source

Select the peripheral interrupt that triggers a DMA burst for the specified channel.

Select SEQ1INT or SEQ2INT to configure the ADC interrupt as interrupt source.

Select XINT1, XINT2, or XINT13 to configure GPIO pin 0 to 31 as an external interrupt source. Select XINT3 to XINT7 to configure GPIO pin 32 to 63 as an external interrupt source.

For more information about configuring XINT, consult the following references:

- *TMS320x2833x, 2823x External Interface (XINTF) User's Guide*, Literature Number: SPRU949, available on the TI Web site.
- *TMS320x2833x System Control and Interrupts*, Literature Number: SPRUFB0, available on the TI Web site.
- *TMS320F28M3x*, Literature Number: SPRUH22F/ SPRUHE8B available on the TI Web site.
- The and block reference sections.

Drop-down menu items from TINT0 to MREVTB may require manual configuration.

Select ePWM1SOCA through ePWM6SOCB to configure the ePWM interrupt as an interrupt source. Note that not all revisions of the TMS320F2833x silicon provide ePWM interrupts as sources for DMA transfers. For more information about silicon revisions consult the following reference:

TMS320x2833x, 2823x Silicon Errata/ TMS320F28M3x, Literature Number: SPRZ272/ SPRUH22F/ SPRUHE8B, available on the TI Web site.

The **Interrupt source** parameter corresponds to bit 4-0 (PERINTSEL) in the Mode Register (MODE).

External pin(GPIO)

When you set **Interrupt source** is set to an external interface (XINT[#]), specify the GPIO pin number from which the interrupt originates.

This parameter corresponds to the GPIO XINTn, XNMI Interrupt Select (GPIOXINTnSEL, GPIOXNMISEL) Registers.

For more information, consult the *TMS320x2833x / TMS320F28M3x System Control and Interrupts Reference Guide*, Literature Number SPRUFB0/ SPRUH22F/ SPRUHE8B available from the TI Web site.

SRC wrap

Specify the number of bursts before returning the current source address pointer to the **Source Begin Address** value. To disable wrapping, enter a value for **SRC wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (SRC_WRAP_SIZE) in the Source Wrap Size Register (SRC_WRAP_SIZE).

DST wrap

Specify the number of bursts before returning the current destination address pointer to the **Destination Begin Address** value. To disable wrapping, enter a value for **DST wrap** that is greater than the **Transfer** value.

This parameter corresponds to bits 15-0 (DST_WRAP_SIZE) in the Destination Wrap Size Register (DST_WRAP_SIZE).

SRC Begin address

Set the starting address for the current source address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **SRC wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Source Begin Register (SRC_BEG_ADDR).

DST Begin address

Set the starting address for the current destination address pointer. The DMA module points to this address at the beginning of a transfer and returns to it as specified by the **DST wrap** parameter.

This parameter corresponds to bits 21-0 (BEGADDR) in the Active Destination Begin Register (DST_BEG_ADDR).

Burst

Specify the number of 16-bit words in a burst, from 1 to 32. The DMA module must complete a burst before it can service the next channel.

Set the **Burst** value for the peripheral the DMA module is servicing. For the ADC, the value equals the number of ADC registers used, up to 16. For multichannel buffered serial ports (McBSP), which lack FIFOs, the value is 1.

For RAM, the value can range from 1 to 32.

This parameter corresponds to bits 4-0 (BURSTSIZE) in the Burst Size Register (BURST_SIZE).

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

Transfer

Specify the number of bursts in a transfer, from 1 to 65536.

This parameter corresponds to bits 15-0 (TRANSFERSIZE) in the Transfer Size Register (TRANSFER_SIZE).

SRC Burst step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually.

Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (SRCBURSTSTEP) in the Source Burst Step Size Register (SRC_BURST_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

DST Burst step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next burst. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Burst step** to 0. For example, because McBSP does not use FIFO, configure DMA to maintain the sequence of the McBSP data by moving each word of the data individually. Accordingly, when you use DMA to transmit or receive McBSP data, set **Burst size** to 1 word and **Burst step** to 0.

This parameter corresponds to bits 15-0 (DSTBURSTSTEP) in the Destination Burst Step Size Register (DST_BURST_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

SRC Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (SRCTRANSFERSTEP) Source Transfer Step Size Register (SRC_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **SRC wrap** is enabled) the corresponding source **Transfer step** does not alter the results.

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

DST Transfer step

Set the number of 16-bit words by which to increment or decrement the current address pointer before the next transfer. Enter a value from –4096 (decrement) to 4095 (increment).

To disable incrementing or decrementing the address pointer, set **Transfer step** to 0.

This parameter corresponds to bits 15-0 (DSTTRANSFERSTEP) Destination Transfer Step Size Register (DST_TRANSFER_STEP).

If DMA is configured to perform memory wrapping (if **DST wrap** is enabled) the corresponding destination **Transfer step** does not alter the results.

Note This parameter is based on a 16-bit word size. If you set **Data size** to 32 bit, double the value of this parameter.

SRC Wrap step

Set the number of 16-bit words by which to increment or decrement the SRC_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Source Wrap Step Size Registers (SRC_WRAP_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to **32 bit**, double the value of this parameter.

DST Wrap step

Set the number of 16-bit words by which to increment or decrement the DST_BEG_ADDR address pointer when a wrap event occurs. Enter a value from –4096 (decrement) to 4095 (increment).

This parameter corresponds to bits 15-0 (WRAPSTEP) in the Destination Wrap Step Size Registers (DST_WRAP_STEP).

Note This parameter is based on a 16-bit word size. If you set **Data size** to **32 bit**, double the value of this parameter.

Generate interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via the PIE at the beginning or end of a data transfer.

This parameter corresponds to bit 15 (CHINTE) and bit 9 (CHINTMODE) in the Mode Register (MODE).

Enable one shot mode

Enable this parameter to have the DMA channel complete an entire *transfer* in response to an interrupt event trigger.

This option allows a single DMA channel and peripheral to dominate resources, and may streamline processing, but it also creates the potential for resource conflicts and delays.

Disable this parameter to have DMA complete one *burst* per channel per interrupt.

Sync enable

When **Interrupt source** is set to SEQ1INT, enable this parameter to reset the DMA wrap counter when it receives the ADCSYNC signal from SEQ1INT. This way, the wrap counter and the ADC channels remain synchronized with each other.

If **Interrupt source** is not set to SEQ1INT, **Sync enable** does not alter the results.

This parameter corresponds to bit 12 (SYNCE) of the Mode Register (MODE).

Enable continuous mode

Select this parameter to leave the DMA channel enabled upon completing a transfer. The channel will wait for the next interrupt event trigger.

Clear this parameter to disable the DMA channel upon completing a transfer. The DMA module disables the DMA channel by clearing the RUNSTS bit in the CONTROL register when it completes the transfer. To use the channel again, first reset the RUN bit in the CONTROL register.

Enable DST sync mode

When **Sync enable** is enabled, enabling this parameter resets the destination wrap counter (DST_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

Disabling this parameter resets the source wrap counter (SCR_WRAP_COUNT) when the DMA module receives the SEQ1INT interrupt/ADCSYNC signal.

This parameter is associated with bit 13 (SYNCSEL) in the Mode Register (MODE).

Set channel 1 to highest priority

This parameter is only available for DMA_ch1.

Enable this setting when DMA channel 1 is configured to handle high-bandwidth data, such as ADC data, and the other DMA channels are configured to handle lower-priority data.

When enabled, the DMA module services each enabled channel sequentially until it receives a trigger from channel 1.

Upon receiving the trigger, DMA interrupts its service to the current channel at the end of the current word, services the channel 1 burst that generated the trigger, and then continues servicing the current channel at the beginning of the next word.

Disable this channel to give each DMA channel equal priority, or if DMA channel 1 is the only enabled channel.

When disabled, the DMA module services each enabled channel sequentially.

This parameter corresponds to bit 0 (CH1PRIORITY) in the Priority Control Register 1 (PRIORITYCTRL1).

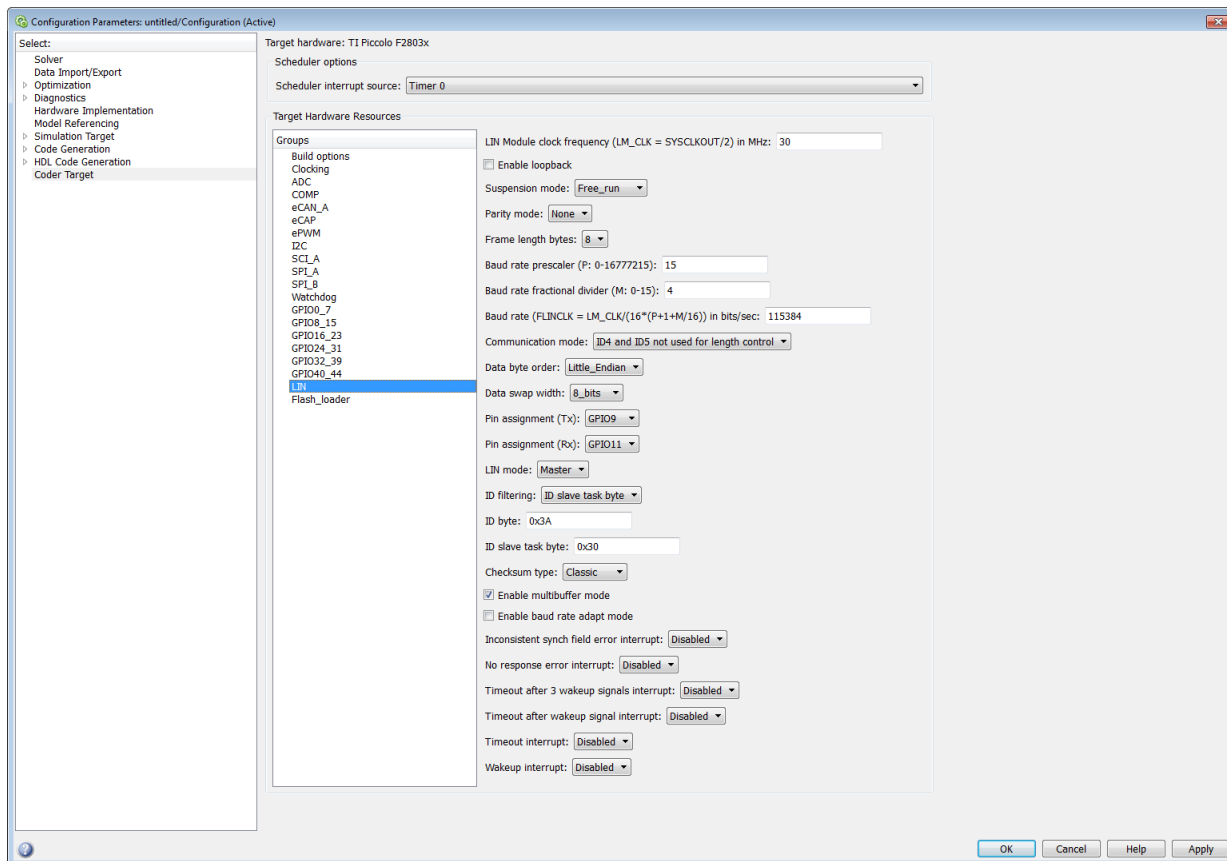
Enable overflow interrupt

Enable this parameter to have the DMA channel send an interrupt to the CPU via PIE if the DMA module receives a peripheral interrupt while a previous interrupt from the same peripheral is waiting to be serviced.

This parameter is typically used for debugging during the development phase of a project.

The **Enable overflow interrupt** parameter corresponds to bit 7 (OVRINTE) of the Mode Register (MODE), and involves the Overflow Flag Bit (OVRFLG) and Peripheral Interrupt Trigger Flag Bit (PERINTFLG).

LIN



For detailed information on the LIN module, see *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2, available at the Texas Instruments Web site.

The following options configure all LIN Transmit and LIN Receive blocks within a model.

LIN Module clock frequency (LM_CLK = SYSCLKOUT/2) in MHz

Displays the frequency of the LIN module clock in MHz.

Enable loopback

To enable LIN loopback testing, select this option. While this option is enabled, the LIN module does the following:

- Internally redirects the LINTX output to the LINRX input.
- Puts the external LINTX pin into high state.
- Puts the external LINRX pin into a high impedance state.

The default is disabled (unchecked).

Suspension mode

Use this option to configure how the LIN state machine behaves while you debug the program on an emulator. If you select **Hard_abort**, entering LIN debug mode halts the transmissions and counters.

The transmissions and counters resume when you exit LIN debug mode. If you select **Free_run**, entering LIN debug mode allows the current transmit and receive functions to complete.

The default is **Free_run**.

Parity mode

Use this option to configure parity checking:

- To disable parity checking, select **None**.
- To enable odd parity checking, select **Odd**.
- To enable even parity checking, select **Even**.

The default is **None**.

In order for **ID parity error interrupt** in the LIN Receive block to generate interrupts, also enable **Parity mode**.

Frame length bytes

Set the number of data bytes in the response field, from 1 to 8 bytes.

The default is 8 bytes.

Baud rate prescaler (P: 0-16777215)

To set the LIN baud rate manually, enter a prescaler value, from 0 to 16777215. Click **Apply** to update the **Baud rate** display.

The default is 15.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

Baud rate fractional divider (M: 0–15)

To set the LIN baud rate manually, enter a fractional divider value, from 0 to 15. Click **Apply** to update the **Baud rate** display.

The default is 4.

For more information, consult the “Baud Rate” topic in the TI document, *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2.

Baud rate (FLINCLK = $LM_CLK/(16*(P+1+M/16))$ in bits/sec

This field displays the baud rate. For more information, see “Setting the LIN baud rate”.

Communication mode

Enable or disable the LIN module from using the ID-field bits ID4 and ID5 for length control.

The default is ID4 and ID5 not used for length control

Data byte order

Set the “endianness” of the LIN message data bytes to `Little_Endian` or `Big_Endian`.

The default is `Little_Endian`.

Data swap width

Select `8_bits` or `16_bits`. If you set **Data byte order** to `Big_Endian`, the only available option for **Data swap width** is `8_bits`.

Pin assignment (Tx)

Map the LINTX output to a specific GPIO pin.

The default is `GPIO9`.

Pin assignment (Rx)

Map the LINRX input to a specific GPIO pin.

The default is `GPIO11`.

LIN mode

Put the LIN module in `Master` or `Slave` mode. The default is `Slave`.

In master mode, the LIN node can transmit queries and commands to slaves. In slave mode, the LIN module responds to queries or commands from a master node.

This option corresponds to the CLK_MASTER field in the SCI Global Control Register (SCIGCR1).

ID filtering

Select which type of mask filtering comparison the LIN module performs, **ID byte** or **ID slave task byte**.

If you select **ID byte**, the module uses the RECID and ID-BYTE fields in the LINID register to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module does not report matches.

If you select **ID slave task**, the module uses the RECID and ID-SlaveTask byte to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module reports matches.

The default is **ID slave task byte**.

ID byte

If you set **ID filtering** to **ID byte**, use this option to set the ID BYTE, also known as the “LIN mode message ID”.

In master mode, the CPU writes this value to initiate a header transmission. In slave mode, the LIN module uses this value to perform message filtering.

The default is 0x3A.

ID slave task byte

If you set **ID filtering** to **ID slave task byte**, use this option to set the ID-SlaveTask BYTE. The LIN node compares this byte with the Received ID and determines whether to send a transmit or receive response.

The default is 0x30.

Checksum type

Use this option to select the type of checksum. If you select **Classic**, the LIN node generates the checksum field from the data fields in the response.

If you select **Enhance**, the LIN node generates the checksum field from both the ID field in the header and data fields in the response. LIN 1.3 supports classic checksums only. LIN 2.0 supports both classic and enhanced checksums.

The default is **Classic**.

Enable multibuffer mode

When you enable (select) this checkbox, the LIN node uses transmit and receive buffers instead of just one register. This setting affects various other LIN registers, such as: checksums, framing errors, transmitter empty flags, receiver ready flags, transmitter ready flags.

The default is enabled (checked).

Enable baud rate adapt mode

The dialog box displays this option when you set **LIN mode** to **Slave**.

If you enable this option, the slave node automatically adjusts its baud rate to match that of the master node. For this feature to work, first set the **Baud rate prescaler** and **Baud rate fractional divider**.

If you disable this option, the LIN module sets a static baud rate based on the **Baud rate prescaler** and **Baud rate fractional divider**.

The default is disabled (unchecked).

Inconsistent synch field error interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

If you enable this option, the slave node generates interrupts when it detects irregularities in the synch field. This option is only relevant if you enable **Enable adapt mode**.

The default is Disabled.

No response error interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

If you enable this option, the LIN module generates an interrupt if it does not receive a complete response from the master node within a timeout period.

The default is Disabled.

Timeout after 3 wakeup signals interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When enabled, the slave node generates an interrupt when it sends three wakeup signals to the master node and does not receive a header in response. (The slave waits 1.5 seconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is having a problem recovering from low-power or sleep mode.

The default is **Disabled**.

Timeout after wakeup signal interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When enabled, the slave node generates an interrupt when it sends a wakeup signal to the master node and does not receive a header in response. (The slave waits 150 milliseconds before sending another series of wakeup signals.)

This interrupt typically indicates the master node is delayed recovering from low-power or sleep mode.

The default is **Disabled**.

Timeout interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When enabled, the slave node generates an interrupt after 4 seconds of inactivity on the LIN bus.

The default is **Disabled**.

Wakeup interrupt

The dialog box displays this option when you set **LIN mode** to **Slave**.

When you enable this option:

- In low-power mode, a LIN slave node generates a wakeup interrupt when it detects the falling edge of a wake-up pulse or a low level on the LINRX pin.
- A LIN slave node that is “awake” generates a wakeup interrupt if it receives a request to enter low-power mode while it is receiving.
- A LIN slave node that is “awake” does not generate a wakeup interrupt if it receives a wakeup pulse.

The default is **Disabled**.

Coder Target Pane

The Coder Target pane is visible when you set the following parameters on the Code Generation pane as follows:

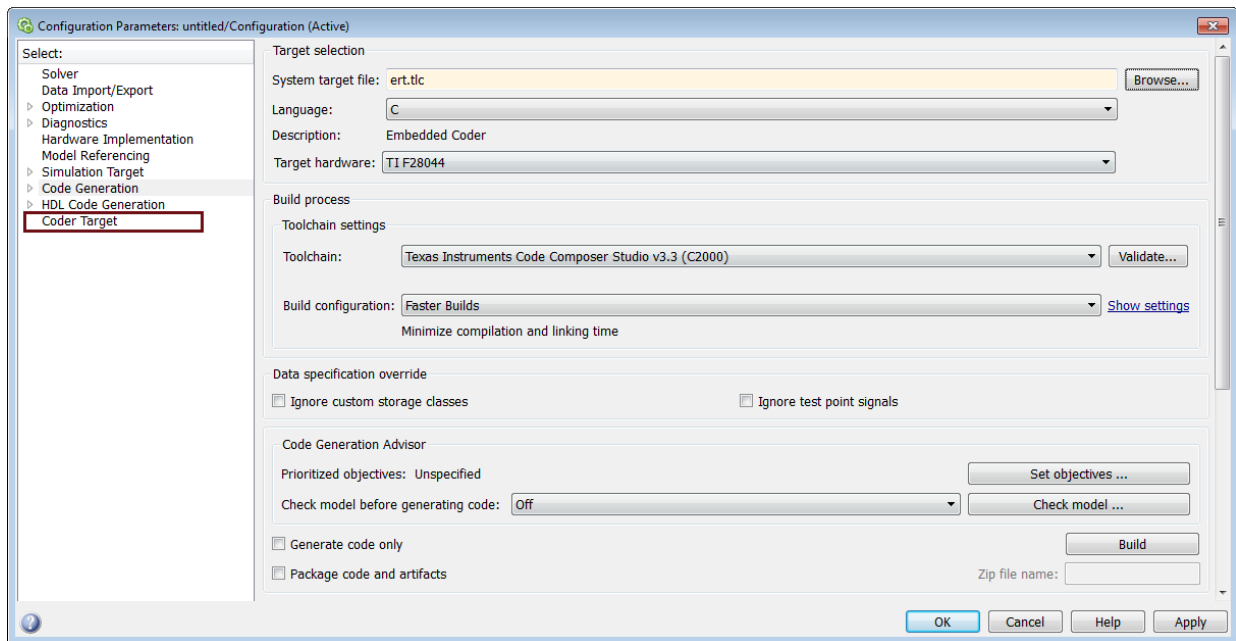
System target file to `ert.tlc`.

Target hardware to a specific type of hardware (not None).

If you are a new user, the `ert.tlc` system target file is the preferred option over the `idelink_ert.tlc`. This option provides a uniform model configuration workflow across the phases of development such as simulation, target prototyping and production code generation.

The features that the `ert.tlc` workflow supports include:

- Production code generation workflow.
- Auto-download and run (when supported with the respective vendor tools).



System target file

Specify the system target file.

Settings

Default: `grt.tlc`

Use the System Target File Browser. Click the **Browse** button, which lets you select a preset target configuration consisting of a system target file, template makefile, and make command. For uniform model configuration workflow, select `ert.tlc`.

Target hardware

Select the target hardware for which to generate code.

The **Coder Target** pane is visible with corresponding peripherals, when you select a target hardware. Changing the **Target hardware** parameter changes the peripherals visible on the **Coder Target** pane.

Toolchain

Based on the target hardware you select, the corresponding tool chain is automatically selected. You can select a different value from the **Toolchain** list.

Note The above parameters settings are specific to **Coder Target** pane. For the other parameter settings on the screen, see “Code Generation Pane: General”.

Coder Target Pane: Texas Instruments Concerto F28M3x (ARM Cortex-M3)

In this section...

“Coder Target Pane Overview” on page 3-270

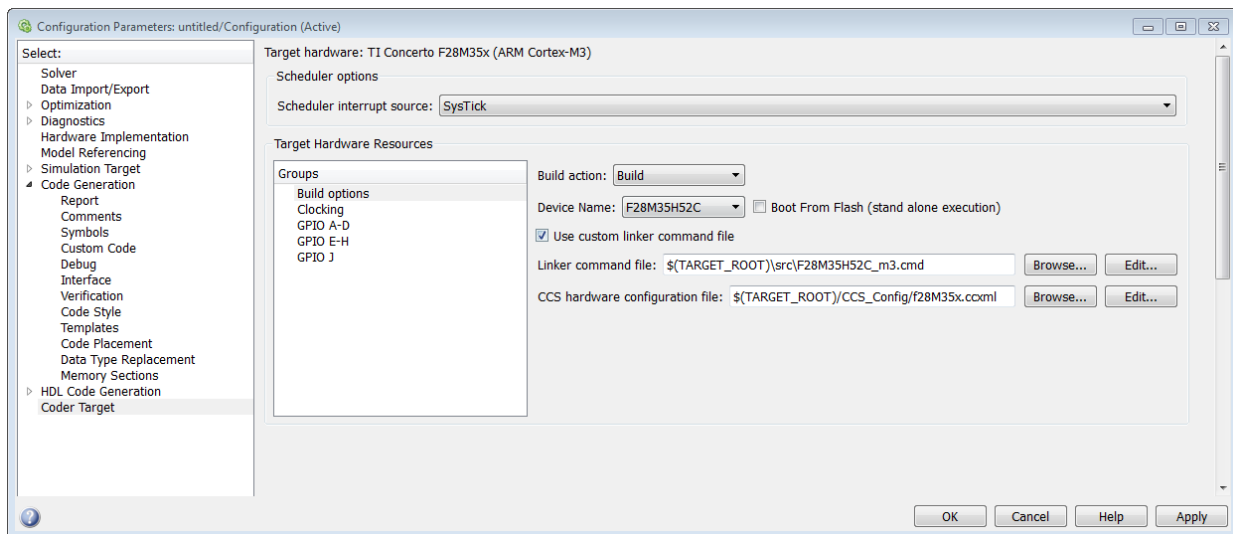
“Coder Target” on page 3-270

“Scheduler options” on page 3-271

“Build options” on page 3-271

“Clocking” on page 3-273

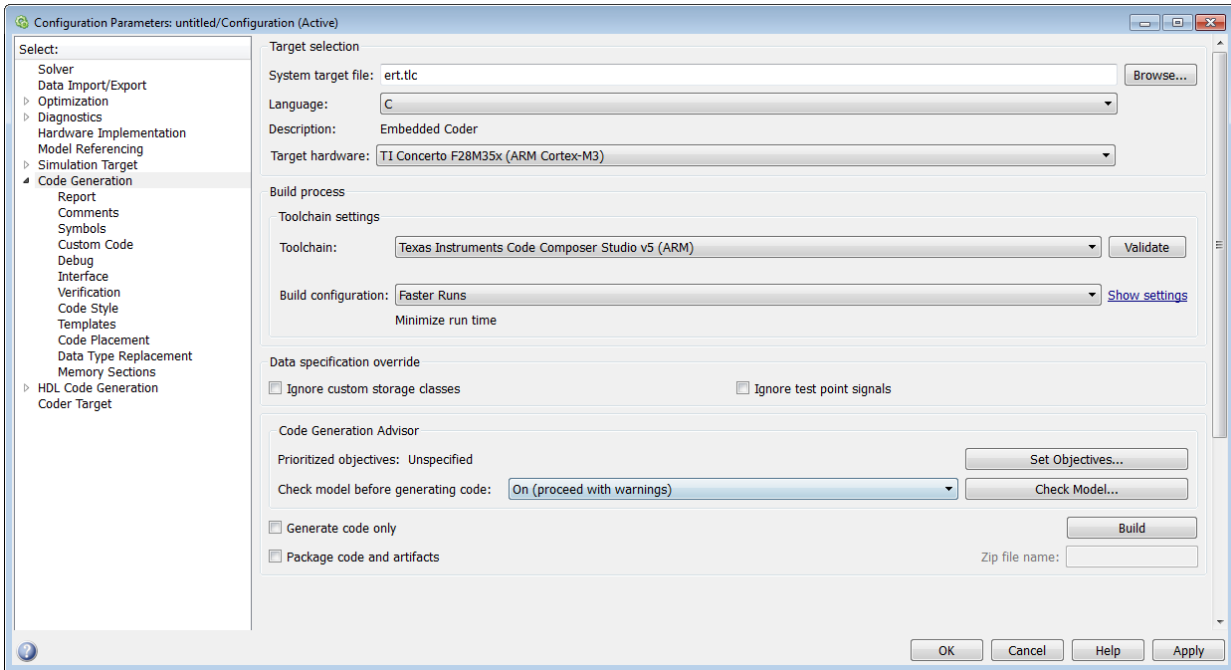
“GPIO A-D” on page 3-275



Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

Coder Target



This Coder Target pane is visible when the following parameters on the Code Generation pane are both set as follows:

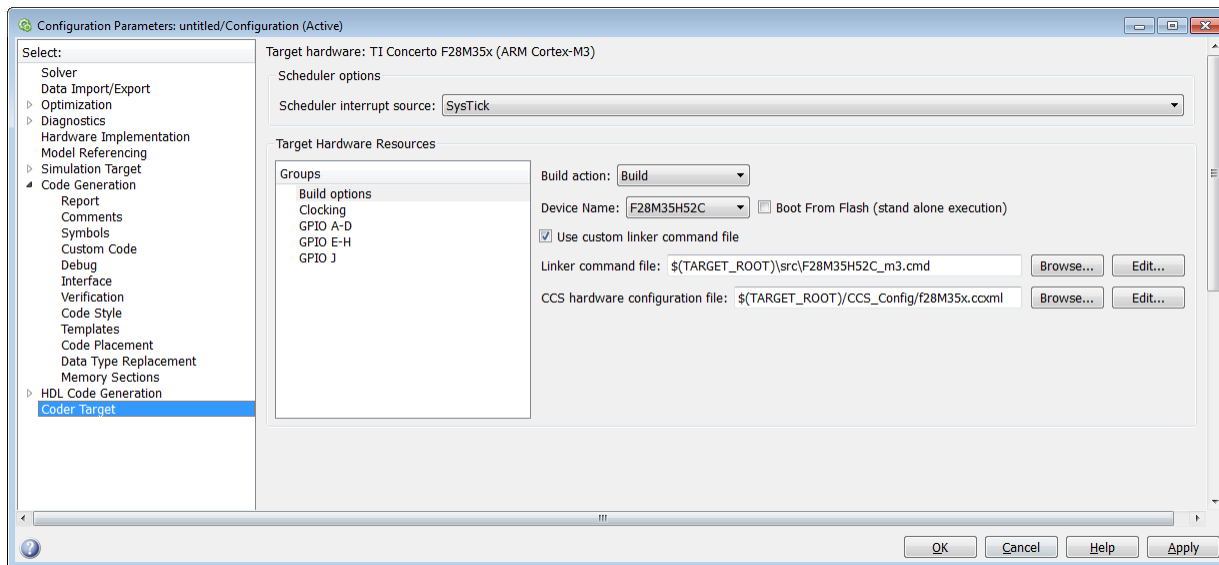
- **System target file** is `ert.tlc`.
- **Target hardware** is TI Concerto F28M35x(ARM Cortex-M3) or TI Concerto F28M36x(ARM Cortex-M3).

Scheduler options

Scheduler interrupt source

Select the source of the scheduler interrupt.

Build options



Use the build options to specify how the build process should take place during code generation.

Build action

The option to specify if you want only 'build' or 'build, load, and run' action during the build process. The **build**, **load** and **run** option is supported only for TI Code Composer Studio v5 (C2000) tool chain.

If you select **build**, **load**, and **run** option, you must provide the required CCS hardware configuration file.

Device name

The option to select a particular device from the selected processor family in the Target hardware parameter on the Code Generation pane.

Boot From Flash (stand alone execution)

The option to specify if the application has to load to the flash. If you do not select this option, the application loads to the RAM.

Use custom linker command file

The option to indicate that the custom linker command file must be used during the build action. Select this option, if you have your own custom linker file, which you

can specify in Linker command file parameter. If you do not select this option, based on the device you have selected, a default custom linker command file will be used.

Linker command file

The path to memory description file that is required during linking. For each family of TI processor selected under 'Target Hardware', one linker command file will be selected automatically.

For different variant of processor, you can select from the 'src' folder inside the Support Package installation path. You can also create custom linker command file and select the file path using **Browse**.

CCS hardware configuration file

The Code Composer Studio file required for downloading the application on the hardware. Select one of the .ccxml files from the folder 'CCS_Config' folder under Support Package installation folder.

Instead, you can also create your own .ccxml file.

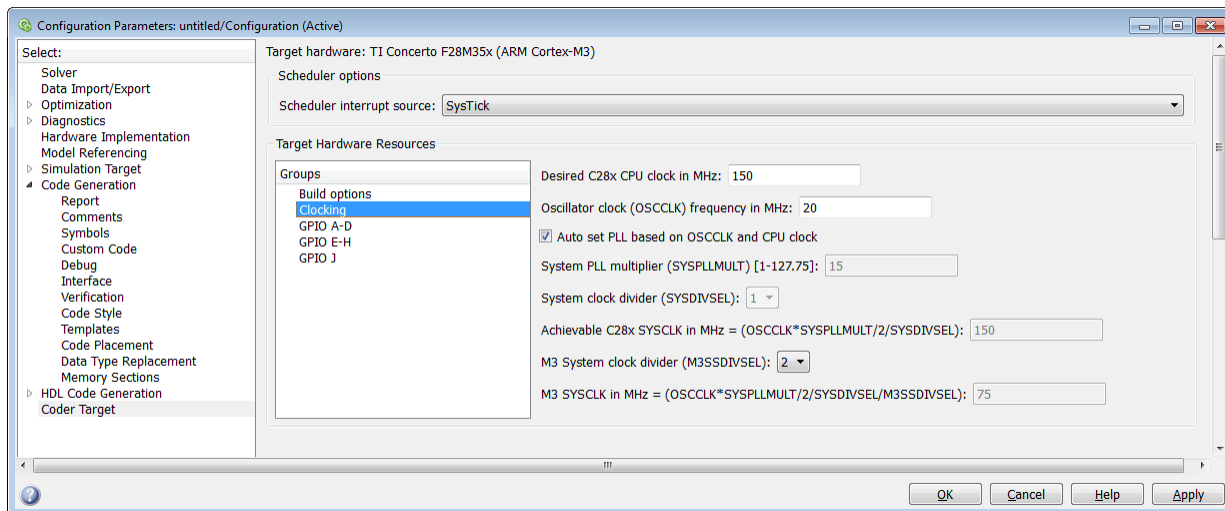
Select the file you created using **Browse**

.

The .ccxml files provided with the support package are as follows:

- f28M35x.ccxml – Texas Instruments XDS100v2 USB Emulator_0
- f28M36x.ccxml – Texas Instruments XDS100v2 USB Emulator_0

Clocking



Desired C28x CPU clock in MHz

Specify the expected C28x CPU clock frequency and match the same in your C28x Model. The C28x CLOCK is the same as PLLSYSCLK. The M3 Clock is a factor of M3SSDIVSEL divided by the PLLSYSCLK.

Oscillator clock (OSCCLK) frequency in MHz

Specify the frequency of the crystal oscillator used in the board. In case of Concerto the crystal oscillator is external to the processor.

Auto set PLL based on OSCCLK and CPU clock

The option that helps you to set the PLL control register value automatically. When you select this check box, the values in the SYSPLLMULT, SYSDIVSEL, and the Achievable C28x SYSCLK in MHz parameters are automatically calculated based on the **Desired C28x CPU Clock** value entered on the Board.

System PLL multiplier (SYSPLLMULT)[1–127.75]

Specify the system PLL multiplier. You can specify a value in this parameter if **Auto set PLL based on OSCCLK and CPU clock** is not selected. The PLL multiplier is a 9 bit field with 7 bits of the SYSPLLMULT register comprising of the integer portion and the remaining 2 bits for the fractional portion. You can enter a value in the range between 0 to 127.75 with multiples of 0.25 for fractional portion of the value.

System clock divider (SYSDIVSEL)

If you select the **Auto set PLL based on OSCCLK and CPU clock** check box, the auto calculated clock divider value achieves the specified CPU Clock value based on the Oscillator clock frequency. Otherwise, you can select a value for Clock divider (SYSDIVSEL).

Achievable C28x SYSCLK in MHz = (OSCCLK * SYSPLLMULT/ 2/ SYSDIVSEL)

The auto calculated feedback value that matches most closely to the desired CPU Clock value on the board, based on the values of OSCCLK, SYSPLLMULT, and the SYSDIVSEL.

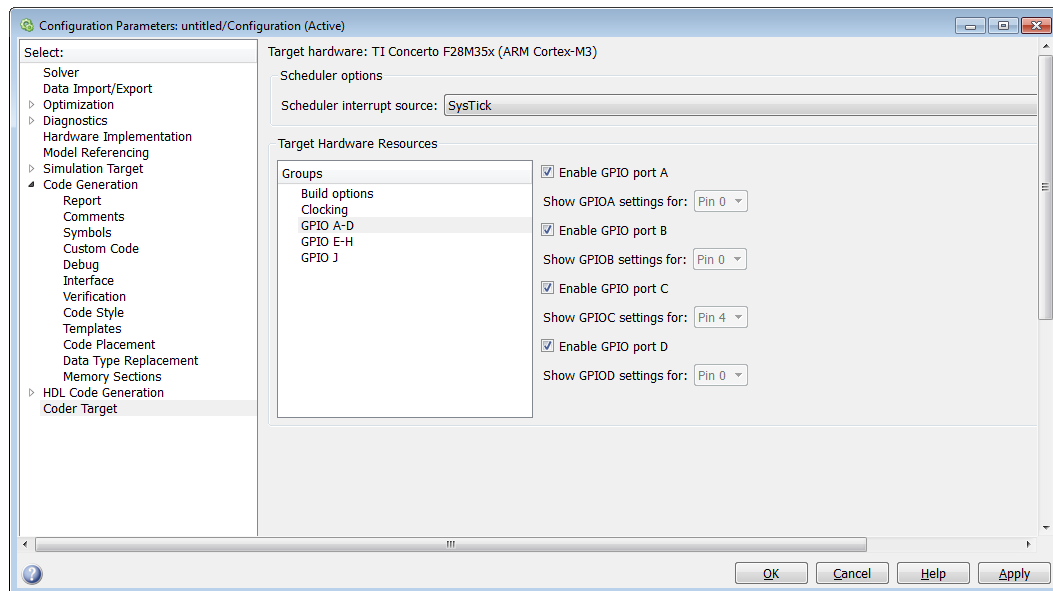
M3 System clock divider (M3SSDIVSEL)

Select a value from the options for M3 system clock divider. The C28 CLKIN clock is divided by the selected value to generate the M3 CPU clock.

M3 SYSCLK in MHz = (OSCCLK * SYSPLLMULT/ 2/ SYSDIVSEL/ M3SSDIVSEL)

This is the achievable M3 system clock frequency. This is calculated based on the values of OSCCLK, SYSPLLMULT, SYSDIVSEL, and M3SSDIVSEL.

GPIO A-D



Enable GPIO port A

Select this option to enable GPIO port A.

Show GPIOA settings for

Select GPIO pins from port A for which you want to set the CPU core and the pin type.

Select the CPU core which controls Pin #

Select the CPU core which controls Pin #

Select the CPU core for the selected GPIO pin.

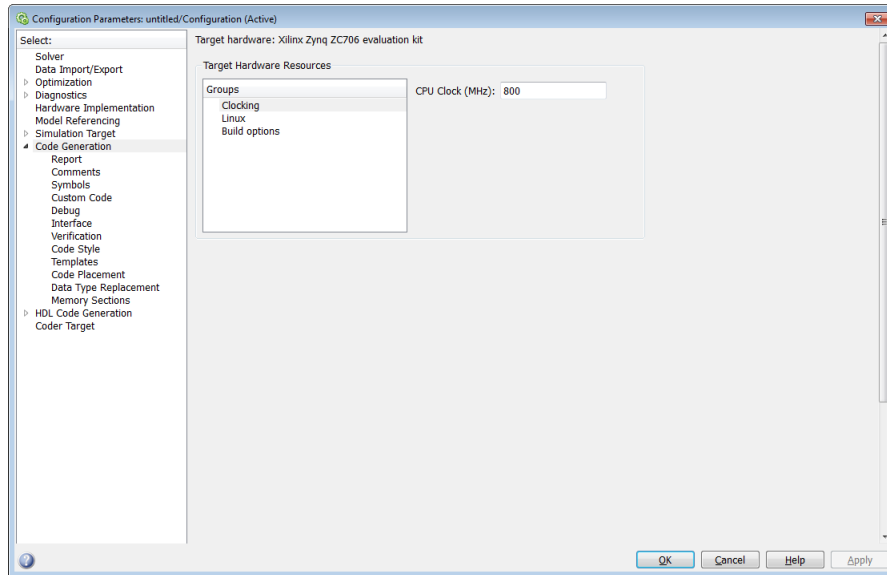
- Auto detect M3 usage, otherwise set to C28x — This default option detects, if you have used the selected GPIO pin for the M3 block in your model.
- M3 — Select this option to assign the GPIO pin to the M3 CPU.
- C28x — Select this option to assign the GPIO pin to the C28x CPU.

Select the pin type for Pin 0

Select the pull-up and the open-drain options for the selected GPIO pin.

Note The above parameter descriptions are also applicable for the other GPIO ports.

Coder Target Pane: Xilinx Zynq ZC702/ZC706 Evaluation Kits, ZedBoard



In this section...

“Coder Target Pane Overview” on page 3-277

“Coder Target” on page 3-277

“Clocking” on page 3-278

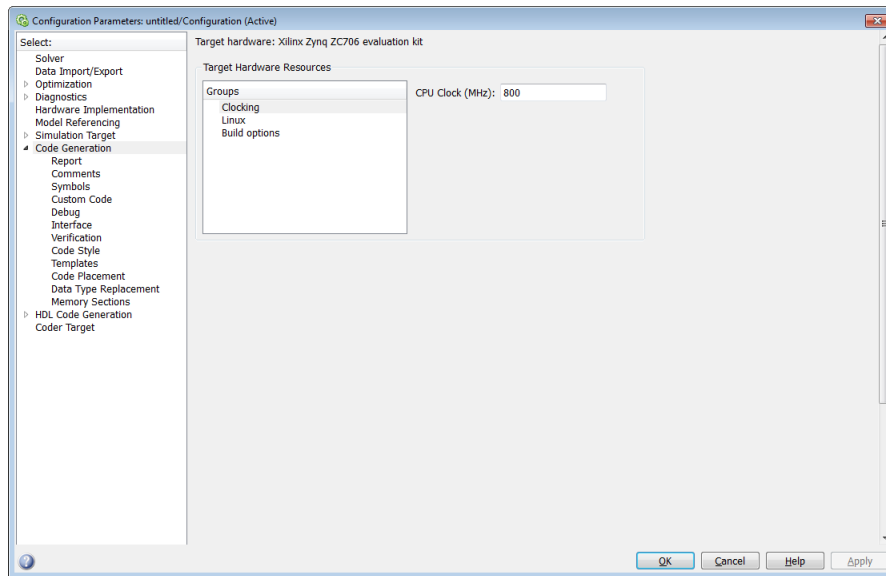
“Linux” on page 3-278

“Build Options” on page 3-278

Coder Target Pane Overview

Configure the parameters for properties of the physical hardware, such as peripherals.

Coder Target



This Coder Target pane is visible when the following parameters are on the Code Generation pane are both set as follows:

- **System target file** is `ert.tlc`
- **Target hardware** is Xilinx Zynq ZC702 evaluation kit

Clocking

CPU Clock (MHz)

Specify the CPU clock frequency of the real ARM Cortex processor on the target hardware.

Linux

Base Rate Task Priority

This parameter sets the static priority of the base rate task. By default, the priority is 40.

Build Options

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: Build, load and run

Build, load and run

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.
- 3 Loads the executable and libraries into the QEMU emulator.
- 4 Runs the executable in the QEMU emulator.

Build

With this option, pressing Ctrl+B or clicking Build Model:

- 1 Generates code from the model.
- 2 Compiles and links the code into an executable with libraries.

This option does not load and run the executable in the QEMU emulator.

Command-Line Information

Parameter: buildAction

Type: string

Value: Build | Build_load_and_run |

Default: Build_load_and_run

Recommended Settings

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

See Also

For more information, refer to the “Code Generation Pane: Coder Target” topic.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic.

Parameter Reference

In this section...

“Recommended Settings Summary” on page 3-281

“Parameter Command-Line Information Summary” on page 3-293

Recommended Settings Summary

The following table summarizes the impact of each Embedded Coder configuration parameter on debugging, traceability, efficiency, and safety considerations, and indicates the factory default configuration settings for the ERT target. The Simulink Coder configuration parameters are documented in “Recommended Settings Summary”. For additional details, click the links in the Configuration Parameter column.

Mapping of Application Requirements to the Optimization Pane : General tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Application lifespan (days)”	No impact	No impact	Optimal finite value	inf	1(ERT targets)
“Optimize using the specified minimum and maximum values”	Off	Off	On	Off	Off
“Remove root level I/O zero initialization”	No impact	No impact	On (GUI) off (command line) (execution, ROM), No impact (RAM)	Off	Off
“Remove internal data zero initialization”	No impact	No impact	On (GUI) off (command line) (execution,	Off	Off

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
			ROM), No impact (RAM)		
“Optimize initialization code for model reference”	No impact	No impact	On (execution, ROM), No impact (RAM)	No impact	On
“Remove code that protects against division arithmetic exceptions”	No impact	No impact	On	Off	Off

Mapping of Application Requirements to the Optimization Pane: Signals and Parameters tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Simplify array indexing”	No impact	No impact	No impact	No impact	Off
“Pack Boolean data into bitfields”	No impact	No Impact	Off (execution, ROM), On (RAM)	No impact	Off
“Bitfield declarator type specifier”	No impact	No impact	Target dependent	No impact	uint_T
“Pass reusable subsystem outputs as”	No impact	No impact	No impact (execution), Structure reference (ROM), Individual arguments (RAM)	No impact	Structure reference

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Optimize global data access”	No impact	No impact	None (execution, ROM, RAM), Use global to hold temporary results (ROM, RAM, execution) Minimize global data access (ROM, RAM, execution)	No impact	None
“Parameter structure”	No impact	Hierarchical	Non-Hierarchical	No impact	Hierarchical

Mapping of Application Requirements to the Code Generation Pane

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Ignore custom storage classes”	No impact	No impact	No impact	No impact	Off
“Ignore test point signals”	Off	No impact	On	No impact	Off

Mapping of Application Requirements to the Code Generation 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

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Generate model Web view”	No impact	No impact	No impact	No impact	Off
“Eliminated / virtual blocks”	On	On	No impact	On	Off
“Traceable Simulink blocks”	On	On	No impact	On	Off
“Traceable Stateflow objects”	On	On	No impact	On	Off
“Traceable MATLAB functions”	On	On	No impact	On	Off
“Static code metrics”	No impact	No impact	No impact	No impact	Off
“Summarize which blocks triggered code replacements”	No impact	No impact	No impact	No impact	Off

Mapping of Application Requirements to the Code Generation 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”	On	On	No impact	No impact	Off

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
(MPT objects only)”					
“Custom comments function”	Valid file name	Valid file name	No impact	No impact	' '
“Stateflow object descriptions”	On	On	No impact	No impact	Off
“Requirements in block comments”	On	On	No impact	On	Off

Mapping of Application Requirements to the Code Generation Pane: Symbols Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Global variables”	No impact	Valid combination of tokens	No impact	\$R\$N\$M	\$R\$N\$M
“Global types”	No impact	Valid combination of tokens	No impact	\$N\$R\$M_T	&N\$R\$M_T
“Field name of global types”	No impact	Valid combination of tokens	No impact	\$N\$M	\$N\$M
“Subsystem methods”	No impact	Valid combination of tokens	No impact	\$R\$N\$M\$F	\$R\$N\$M\$F
“Subsystem method arguments”	No impact	Valid combination of tokens	No impact	rtu_\$N\$M or rty_\$N\$M	rt\$I\$N\$M
“Local temporary variables”	No impact	Valid combination of tokens	No impact	\$N\$M	\$N\$M

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Local block output variables”	No impact	Valid combination of tokens	No impact	rtb_\$\$M	rtb_\$\$M
“Constant macros”	No impact	Valid combination of tokens	No impact	\$\$R\$\$M	\$\$R\$\$M
“Shared utilities”	No impact	Valid combination of tokens	No impact	\$\$N\$\$C	\$\$N\$\$C
“Minimum mangle length”	No impact	1	No impact	No impact	1
“System-generated identifiers”	No impact	No impact	No impact	No impact	Shortened
“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
“MATLAB function”	No impact	No impact	No impact	No impact	' '

Mapping of Application Requirements to the Code Generation Pane: Interface Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Support: floating-point numbers”	No impact	No impact	Off (GUI), 'on' (command-	No impact	On (GUI), 'off' (command-line)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
			line) for integer only		
“Support complex numbers”	No impact	No impact	Off for real only	No impact	On
“Support absolute time”	No impact	No impact	Off	Off	On
“Support continuous time”	No impact	No impact	Off (execution, ROM), No impact (RAM)	Off	Off
“Support non-inlined S-functions”	No impact	No impact	Off	Off	Off
“Support variable-size signals”	No impact	No impact	Off	Off	Off
“Multiword type definitions”	No impact	No impact	No impact	Use default	System defined
“Maximum word length”	No impact	No impact	No impact	Use default	256
“Pass root-level I/O as”	No impact	No impact	No impact	No impact	Individual arguments
“Use dynamic memory allocation for model initialization”	No impact	No impact	No impact	Off	Off
“Terminate function required”	No impact	No impact	Off (execution, ROM), No impact (RAM)	Off	On

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Generate preprocessor conditionals”	No impact	No impact	No impact	No impact	Use local settings
“Suppress error status in real-time model data structure”	Off	No impact	On	On	Off
“Combine signal/state structures”	Off	No impact	No impact	On	No impact
“Block parameter visibility”	No impact	No impact	No impact	protected	private
“Internal data visibility”	No impact	No impact	No impact	protected	private
“Block parameter access”	Inlined method	Inlined method	Inlined method	None	None
“Internal data access”	Inlined method	Inlined method	Inlined method	None	None
“External I/O access”	Inlined method	Inlined method	Inlined method	None	None
“Generate destructor”	No impact	No impact	No impact	Off	On
“Use dynamic memory allocation for model block instantiation”	No impact	No impact	On	Off	Off

Mapping of Application Requirements to the Code Generation Pane: Verification Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
“Code coverage tool”	BullseyeCove or LDRA Testbed	BullseyeCove or LDRA Testbed	None (code coverage off)	None (code coverage off)	None (code coverage off)
“Create block”	On	No impact	No impact	No impact	Off
“Enable portable word sizes”	On	On	Off	No impact	Off
“Enable source-level debugging for SIL”	On	On	Off	No impact	Off
“Measure task execution time”	On	On	Off	No impact	Off
“Measure function execution times”	On	On	Off	No impact	Off
“Save options”	All measurement and analysis data	All measurement and analysis data	Summary data only	No impact	Summary data only
“Workspace variable”	No impact	Valid MATLAB variable name	No impact	No impact	Off

Mapping of Application Requirements to the Code Generation 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)	Maximum (Specify precedence)	Nominal (Optimize for readability)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
			for precedence)	with parentheses)	
Preserve operand order in expression	On	On	Off	On	Off
Preserve condition expression in if statement	On	On	Off	On	Off
Convert if-elseif-else patterns to switch-case statements	No impact	Off	On (execution, ROM), No impact (RAM)	No impact	Off
Preserve extern keyword in function declarations	No impact	No impact	No impact	No impact	On
Suppress generation of default cases for Stateflow switch statements if unreachable	No impact	On	On (execution, ROM), No impact (RAM)	Off	Off
Casting Modes	Nominal	Nominal	Nominal	Standards Compliant	Nominal
Indent style	K&R	K&R	K&R	K&R	K&R
Indent size	2	2	2	2	2

Mapping of Application Requirements to the Code Generation 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 Code Generation Pane: Code Placement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Data definition	No impact	Valid value	No impact	No impact	Auto
Data definition filename	No impact	Valid value	No impact	No impact	global.c

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Data declaration	No impact	Valid value	No impact	No impact	Auto
Data declaration filename	No impact	Valid value	No impact	No impact	global.h
#include file delimiter	No impact	Valid value	No impact	No impact	off
#include file delimiter	No impact	Valid value	No impact	No impact	Auto
Signal display level	No impact	Valid integer	No impact	No impact	10
Parameter tune level	No impact	Valid integer	No impact	No impact	10
File packaging format	No impact	No impact	No impact	No impact	Modular

Mapping of Application Requirements to the Code Generation Pane: Data Type Replacement Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Replace data type names in the generated code	No impact	On	No impact	No impact	Off
Replacement Name	No impact	Valid string	No impact	' '	' '

Mapping of Application Requirements to the Code Generation 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

Configuration Parameter	Debugging	Traceability	Efficiency	Safety precaution	Factory Default
Execution	No impact	No impact	No impact	No impact	Default
Shared utility	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	No package selected.

Parameter Command-Line Information Summary

The following tables list 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.

Use the `get_param` and `set_param` commands to retrieve and set the values of the parameters on the MATLAB command line or programmatically in scripts. The Configuration Wizard also provides buttons and scripts for customizing code generation.

Note Parameters that are specific to the ERT target or targets based on the ERT target, Stateflow, or the Fixed-Point Designer product are marked with (ERT), (Stateflow), and (Fixed-Point Designer), 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: General tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
NoFixptDivByZeroProtection (ERT) (Fixed-Point Designer) off , on	Remove code that protects against division arithmetic exceptions	Suppress generation of code that guards against division by zero for fixed-point data.
OptimizeModelRefInitCode (ERT) off , on	Optimize initialization code for model reference	<p>Suppresses generation of initialization code for blocks that have states unless the blocks are in a system that can reset its states, such as an enabled subsystem. This results in more efficient code.</p> <p>The following restrictions apply to using the Optimize initialization code for model reference parameter. However, these restrictions do not apply to a Model block that references a function-call model.</p> <ul style="list-style-type: none"> • In a subsystem that resets states, do not include a Model block that references a model that has this parameter set to on. For example, in an enabled subsystem with the States when enabling block parameter set to reset, do not include a Model block that references a model that has the Optimize initialization code

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		<p>for model reference parameter set to on.</p> <ul style="list-style-type: none"> If you set the Optimize initialization code for model reference parameter to off in a model that includes a Model block, do not set the Optimize initialization code for model reference parameter for the Model block to on.
UseSpecifiedMinMax (ERT) <i>string</i> - off , on	Optimize using the specified minimum and maximum values	Use the specified minimum and maximum values, such as block Output minimum and Output maximum, to optimize generated code
ZeroExternalMemoryAtStartu (ERT) off , on	Remove root level I/O zero initialization	Suppress code that initializes root-level I/O data structures to zero.
ZeroInternalMemoryAtStartu (ERT) off , on	Remove internal data zero initialization	Suppress code that initializes global data structures (for example, block I/O data structures) to zero.

Command-Line Information: Optimization Pane: Signals and Parameters tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
InlinedParameterPlacement (ERT) Hierarchical , NonHierarchical	Parameter structure	Specify how generated code stores global (tunable) parameters. Specify NonHierarchical to trade off modularity for efficiency.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
BooleansAsBitfields (ERT) off , on	Pack Boolean data into bitfields	Specify how generated code stores Boolean signals. If selected, Boolean signals are stored into one-bit bitfields in global block I/O structures or DWork vectors.
BitfieldContainerType (ERT) uint_T , uchar_T	Bitfield declarator type specifier	Specify the bitfield type when using the optimization to pack boolean data into bitfields.
GlobalVariableUsage (ERT) None , Use global to hold temporary results, Minimize global data access	Optimize global data access	Specify global data usage optimization.
StrengthReduction (ERT) off , on	Simplify array indexing	Suppress generation of code that replaces multiply operations when accessing arrays in a loop.
PassReuseOutputArgsAs (ERT) Structure reference , Individual arguments	Pass reusable subsystem output as	Specify how a reusable subsystem passes outputs. Specify Individual arguments for efficiency.

Command-Line Information: Optimization Pane: Stateflow tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
DataBitsets (Stateflow) off , on	Use bitsets for storing Boolean data	Use bit sets for storing Boolean data.
StateBitsets (Stateflow) off , on	Use bitsets for storing state configuration	Use bit sets for storing state configuration.

Command-Line Information: Code Generation Pane: General Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
IgnoreCustomStorageClasses (ERT) <i>string</i> - off , on	Code Generation > General > Ignore custom storage classes	Treat custom storage classes as 'Auto'.
IgnoreTestpoints (ERT) <i>string</i> - off , on	Code Generation > General > Ignore test point signals	Specify allocation of memory buffers for test points.

Command-Line Information: Code Generation Pane: Report Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
GenerateTraceInfo (ERT) <i>string</i> - off , on	Code Generation > Report > Model-to-code	Includes model-to-code traceability support in the generated HTML report.
IncludeHyperlinkInReport (ERT) <i>string</i> - off , on	Code Generation > Report > Code-to-model	Link code segments to the corresponding object in the model. This option increases code generation time for large models.
GenerateWebview (ERT) <i>string</i> - off , on	Code Generation > Report > Generate model Web view	Include the model Web view in the code generation report.
GenerateTraceReport (ERT) <i>string</i> - off , on	Code Generation > Report > Eliminated / virtual blocks	Include summary of eliminated and virtual blocks in Code Generation report.
GenerateTraceReportSl (ERT) <i>string</i> - off , on	Code Generation > Report > Traceable Simulink blocks	Include summary of Simulink blocks in Code Generation report.
GenerateTraceReportSf (ERT) <i>string</i> - off , on	Code Generation > Report > Traceable Stateflow objects	Include summary of Stateflow objects in Code Generation report.
GenerateTraceReportEm1 (ERT) <i>string</i> - off , on	Code Generation > Report > Traceable MATLAB functions	Include summary of MATLAB functions in Code Generation report.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
GenerateCodeMetricsReport (ERT) <i>string</i> - off , on	Code Generation > Report > Static code metrics	Include static code metrics report in the code generation report.
GenerateCodeReplacementRep (ERT) <i>string</i> - off , on	Code Generation > Report > Summarize which blocks triggered code replacements	Include code replacement report summarizing replacement functions used and their associated blocks in the code generation report.

Command-Line Information: Code Generation Pane: Comments Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomCommentsFcn (ERT) <i>string</i> - ''	Code Generation > Comments > Custom comments function	Specify the filename of the MATLAB or TLC function that adds the custom comment.
EnableCustomComments (ERT) <i>string</i> - off , on	Code Generation > Comments > Custom comments (MPT objects only)	Add a comment above a signal's or parameter's identifier in the generated file.
InsertBlockDesc (ERT) <i>string</i> - off , on	Code Generation > 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) <i>string</i> - off , on	Code Generation > Comments > Requirements in block comments	Include specified requirements in the generated code as a comment.
SFDataObjDesc (ERT) <i>string</i> - off , on	Code Generation > Comments > Stateflow object descriptions	Insert Stateflow object descriptions into the generated code as a comment.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
SimulinkDataObjDesc (ERT) <i>string</i> - off , on	Code Generation > Comments > Simulink data object descriptions	Insert Simulink data object descriptions into the generated code as comments.

Command-Line Information: Code Generation Pane: Symbols Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CustomSymbolStrBlkIO (ERT) <i>string</i> - rtb_ \$N \$M	Code Generation > 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) <i>string</i> - \$R \$N \$M \$F	Code Generation > Symbols > Subsystem methods	Specify a symbol format rule for subsystem methods. The rule can contain valid C identifier characters and the following macros: \$M - Mangle \$R - Root model name \$N - Name of object \$H - System hierarchy number \$F - Subsystem method name
CustomSymbolStrFcnArg(ERT) <i>string</i> - rt \$I \$N \$M	Code Generation > Symbols > Subsystem method arguments	Specify a symbol format rule for subsystem method arguments. The rule can contain valid C identifier characters and the following macros:

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		<p>\$I — u if the argument is an input or y if the argument is an output</p> <p>\$M - Mangle</p> <p>\$N - Name of object</p>
<p>CustomSymbolStrField (ERT) string - \$N\$M</p>	<p>Code Generation > Symbols > Field name of global types</p>	<p>Specify a symbol format rule for field name of global types. The rule can contain valid C identifier characters and the following macros:</p> <p>\$M - Mangle</p> <p>\$N - Name of object</p> <p>\$H - System hierarchy number</p> <p>\$A - Data type acronym</p>
<p>CustomSymbolStrGlobalVar (ERT) string - \$R\$N\$M</p>	<p>Code Generation > Symbols > Global variables</p>	<p>Specify a symbol format rule for global variables. The rule can contain valid C identifier characters and the following macros:</p> <p>\$M - Mangle</p> <p>\$R - Root model name</p> <p>\$N - Name of object</p>
<p>CustomSymbolStrMacro (ERT) string - \$R\$N\$M</p>	<p>Code Generation > Symbols > Constant macros</p>	<p>Specify a symbol format rule for constant macros. The rule can contain valid C identifier characters and the following macros:</p> <p>\$M - Mangle</p> <p>\$R - Root model name</p> <p>\$N - Name of object</p>
<p>CustomSymbolStrTmpVar (ERT) string - \$N\$M</p>	<p>Code Generation > Symbols > Local temporary variables</p>	<p>Specify a symbol format rule for local temporary variables. The rule can contain valid C identifier</p>

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		characters and the following macros: \$M - Mangle \$R - Root model name \$N - Name of object \$A - Data type acronym
CustomSymbolStrType (ERT) string - \$N\$R\$M	Code Generation > 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
CustomSymbolStrUtil (ERT) string - \$N\$C	Code Generation > Symbols > Shared utilities	Specify a symbol format rule for shared utilities. The rule can contain valid C identifier characters and the following macros: \$N - Name of object \$C - Checksum
DefineNamingFcn (ERT) string - ''	Code Generation > Symbols > #define naming > Custom M-function	Specify a custom MATLAB function to control the naming of symbols with #define statements. You can set this parameter only if DefineNamingRule is set to Custom .
DefineNamingRule (ERT) string - None , UpperCase, LowerCase, Custom	Code Generation > Symbols > #define naming	Specify the rule that changes the spelling of #define names.
IncDataTypeInIds (ERT) off , on	Code Generation > Symbol > Include data type acronym in identifiers	Include acronyms that express data types in signal and work vector

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		<p>identifiers. For example, 'rtB.i32_signame' identifies a 32-bit integer block output signal named 'signame'.</p>
<p>IncHierarchyInIds (ERT) off, on</p>	<p>Code Generation > Symbols > Include system hierarchy number in identifiers</p>	<p>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 <code>hilite_system('<S3>')</code> at the MATLAB prompt. The argument specified with <code>hilite_system</code> requires an uppercase S.</p>
<p>InlinedPrmAccess (ERT) string - Literals, Macros</p>	<p>Code Generation > Symbols > Generate scalar inlined parameters as</p>	<p>Specify whether inlined parameters are coded as numeric constants or macros. Specify Macros for more efficient code.</p>
<p>MangleLength (ERT) int - 1</p>	<p>Code Generation > Symbols > Minimum mangle length</p>	<p>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</p>

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		identifier disturbance when you modify the model.
InternalIdentifier (ERT) string - Shortened , Classic	Code Generation > Symbols > System-generated identifiers	Specify whether the code generator uses shorter, more consistent names for system-generated identifiers.
ParamNamingRule (ERT) string - None , UpperCase, LowerCase, Custom	Code Generation > Symbols > Parameter naming	Select a rule that changes spelling of parameter names.
SignalNamingRule (ERT) string - None , UpperCase, LowerCase, Custom	Code Generation > Symbols > Signal naming	Specify a rule the code generator is to use that changes spelling of signal names.

Command-Line Information: Code Generation Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ERTMultiwordLength (ERT) int - 256	Code Generation > Interface > Maximum word length	Specify a maximum word length, in bits, for which the code generation process will generate system-defined multiword types into the file <code>rtwtypes.h</code> . Specifying 0 provides you complete control over type definitions for multiword data types in generated code.
ERTMultiwordTypeDef (ERT) string - System defined , User defined	Code Generation > Interface > Multiword type definitions	Use system-defined or user-defined type definitions for multiword data types in generated code.
GenerateAllocFcn (ERT) string - off , on	Code Generation > Interface > Use dynamic memory allocation for model initialization	Generate a function to dynamically allocate

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		memory (using <code>malloc</code>) for model data structures.
GenerateDestructor (ERT) string - off, on	Code Generation > Interface > Generate destructor	Generate a destructor for the C++ model class.
GenerateExternalIOAccess- Methods (ERT) string - None , Method, Inlined method, Structure-based method, Inlined structure- based method	Code Generation > Interface > External I/O access	Generate access methods for root-level I/O signals for the C++ model class.
GenerateInternalMember- AccessMethods (ERT) string - None , Method, Inlined method	Code Generation > Interface > Internal data access	Generate access methods for internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states for the C++ model class.
GenerateParameterAccess- Methods (ERT) string - None , Method, Inlined method	Code Generation > Interface > Block parameter access	Generate access methods for block parameters for the C++ model class.
GeneratePreprocessor- Conditionals (ERT) string - Use local settings , Enable all, Disable all	Code Generation > Interface > Generate preprocessor conditionals	Generate preprocessor conditionals locally for each Model block containing variants or globally for all Model blocks in a model.
IncludeMdlTerminateFcn (ERT) string - off, on	Code Generation > Interface > Terminate function required	Generate a terminate function for the model.
InternalMemberVisibility (ERT) string - public, private , protected	Code Generation > Interface > Internal data visibility	Generate internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		as public , private , or protected data members of the C++ model class.
ParameterMemberVisibility (ERT) string - public , private , protected	Code Generation > Interface > Block parameter visibility	Generate the block parameter structure as a public , private , or protected data member of the C++ model class.
PurelyIntegerCode (ERT) string - off , on	Code Generation > 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, Part of model data structure	Code Generation > Interface > Pass root-level I/O as	Specify how the generated code passes root-level I/O data into a reusable function.
SupportAbsoluteTime (ERT) string - off , on	Code Generation > 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	Code Generation > Interface > complex numbers	Support complex data types in the generated code.
SupportContinuousTime (ERT) string - off , on	Code Generation > 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.
SupportVariableSizeSignals (ERT)	Code Generation > Interface > variable-size signals	Generate code for models that use variable-size signals.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
string - off , on		
SuppressErrorStatus (ERT) string - off , on	Code Generation > Interface > Suppress error status in real-time model data structure	Remove the error status field of the real-time model data structure to preserve memory. When selected, SupportContinuousTime is cleared.
CombineSignalStateStructs (ERT) string - off , on	Code Generation > Interface > Combine signal/state structures	Combine a model block's signals (global block I/O structure) and discrete states (DWork vector) into a single data structure in the generated code.
UseOperatorNewForModelRefRegistration (ERT) string - off , on	Code Generation > Interface > Use dynamic memory allocation for model block instantiation	For a model containing Model blocks, specify whether generated code should use the operator new , during model object registration, to instantiate objects for referenced models configured with a C++ class interface.

Command-Line Information: Code Generation Pane: Verification Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CoverageTool (ERT) <i>string</i> - None , BullseyeCoverage, LDRA Testbed	Code Generation > Verification > Code coverage tool	Specify a code coverage tool
Tip To access the CoverageTool parameter, type:		

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
<code>covSettings = get_param(gcs, covSettings.CoverageTool)</code>		
CodeExecutionProfileVariable (ERT) string - executionProfile	Code Generation > Verification > Workspace variable	Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.
CodeExecutionProfiling (ERT) string - off , on	Code Generation > Verification > Measure task execution time	Specify whether to collect execution time profiles for tasks in generated code.
CodeProfilingInstrumentation (ERT) string - off , on	Code Generation > Verification > Measure function execution times	Specify whether to collect execution time profiles for functions in code generated from the model.
CodeProfilingSaveOptions (ERT) string - SummaryOnly , AllData	Code Generation > Verification > Save options	Specify whether to save code profiling measurement and analysis data to base workspace.
CreateSILPILBlock (ERT) string - None , SIL, PIL	Code Generation > Verification > Create block	Create SIL or PIL block to allow verification of source or object code generated from subsystem or top-model components.
PortableWordSizes (ERT) string - off , on	Code Generation > Verification > Enable portable word sizes	Specify that model code should be generated with conditional processing macros that allow the same generated source code files to be used both for software-in-the-loop (SIL) testing on the host platform and for production deployment on the target platform.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
SILDebugging (ERT) string - off , on	Code Generation > Verification > Enable source-level debugging for SIL simulations	Enable use of Microsoft Visual Studio® debugger during SIL simulation.

Command-Line Information: Code Generation Pane: Code Style Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CastingMode (ERT) string - Nominal , Standards, Explicit	Code Generation > Code Style > Casting Mode	Control how data types are cast in generated code.
ConvertIfToSwitch (ERT) string - off , on	Code Generation > Code Style > Convert if-elseif-else patterns to switch-case statements	Control whether if-elseif-else decision logic appears in generated code as switch-case statements.
ParenthesesLevel (ERT) string - Minimum, Nominal , Maximum	Code Generation > Code Style > Parentheses Level	Control existence of optional parentheses in generated code.
PreserveExpressionOrder (ERT) string - off , on	Code Generation > Code Style > Preserve operand order in expression	Control reordering of commutable expressions.
PreserveExternInFcnDecls (ERT) string - off, on	Code Generation > Code Style > Preserve extern keyword in function declarations	Control whether extern keyword appears in function declarations with external linkage in the generated code.
PreserveIfCondition (ERT) string - off , on	Code Generation > Code Style > Preserve condition expression in if statement	Control preservation of if statement conditions.
SuppressUnreachableDefaultCases (ERT) string - off , on	Code Generation > Code Style > Suppress generation of default cases for Stateflow switch statements if unreachable	Control whether to generate default cases for switch-case statements in the code for Stateflow charts.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
IndentStyle (ERT) string - K&R , Allman	Code Generation > Code Style > Indent Style	ROUGH DRAFT Control the placement of braces.
IndentSize (ERT) int - 2	Code Generation > Code Style > Indent Size	ROUGH DRAFT Control the number of characters per indent level. Range can be 2–8.

Command-Line Information: Code Generation Pane: Templates Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
ERTCustomFileTemplate (ERT) string - example_file_process.tlc	Code Generation > Templates > File customization template	Specify a TLC callback script for customizing the generated code.
ERTDataHdrFileTemplate (ERT) string - ert_code_template.cgt	Code Generation > Templates > Header file (*.h) template	Specify a template that organizes the generated data .h header files.
ERTDataSrcFileTemplate (ERT) string - ert_code_template.cgt	Code Generation > 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	Code Generation > Templates > Header file (*.h) template	Specify a template that organizes the generated code .h header files.
ERTSrcFileBannerTemplate (ERT) string - ert_code_template.cgt	Code Generation > Templates > Source file (*.c or *.cpp) template	Specify a template that organizes the generated code .c or .cpp source files.
GenerateSampleERTMain (ERT) string - off , on	Code Generation > Templates > Generate an example main program	Generate an example main program that demonstrates how to deploy the generated code. The program is written

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		to the file <code>ert_main.c</code> or <code>ert_main.cpp</code> .
TargetOS (ERT) string - BareBoardExample , VxWorksExample, NativeThreadsExample	Code Generation > Templates > Target operating system	Specify the target operating system for the example main <code>ert_main.c</code> or <code>ert_main.cpp</code> . BareBoardExample is a generic example that does not assume an operating system. VxWorksExample is tailored to the VxWorks ^a real-time operating system. NativeThreadsExample works with threaded code under the native host operating system.

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

Command-Line Information: Code Generation Pane: Code Placement Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
DataDefinitionFile (ERT) string - global.c	Code Generation > Code Placement > Data definition filename	Specify the name of a single separate <code>.c</code> or <code>.cpp</code> file that contains global data definitions.
DataReferenceFile (ERT) string - global.h	Code Generation > Code Placement > Data declaration filename	Specify the name of a single separate <code>.c</code> or <code>.cpp</code> file that contains global data references.
GlobalDataDefinition (ERT) string - Auto , InSourceFile, InSeparateSourceFile	Code Generation > Code Placement > Data definition	Select the <code>.c</code> or <code>.cpp</code> file where variables of global scope are defined.
GlobalDataReference (ERT)	Code Generation > Data Placement > Data declaration	Select the <code>.h</code> file where variables of global

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
string - Auto , InSourceFile, InSeparateHeaderFile		scope are declared (for example, <code>extern real_T globalvar;</code>).
IncludeFileDelimiter (ERT) string - Auto , UseQuote, UseBracket	Code Generation > Code Placement > #include file delimiter	Specify the delimiter to be used for data objects that do not have a delimiter specified in the IncludeFile property.
EnableDataOwnership string - off , on	Code Generation > Code Placement > Use owner from data object for data definition placement	Specify whether the model uses the ownership setting of a data object for data definition in code generation.
ModuleNamingRule (ERT) string - Unspecified , SameAsModel	Code Generation > Code Placement > Use owner from data object for data definition placement	Specify whether the model uses the ownership setting of a data object for data definition in code generation.
ParamTuneLevel (ERT) int - 10	Code Generation > Code Placement > Parameter tune level	Specify whether the code generator is to declare a parameter data object as tunable global data in the generated code.
SignalDisplayLevel (ERT) int - 10	Code Generation > Code Placement > Signal display level	Specify whether the code generator is to declare a signal data object as global data in the generated code.
ERTFilePackagingFormat (ERT) string - Modular , Compact with separate data files, Compact	Code Generation > Code Placement > File Packaging Format	Specify how the code generator organizes the code into files.

Command-Line Information: Code Generation Pane: Data Type Replacement Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
EnableUserReplacementTypes (ERT) string - off , on	Code Generation > Data Type Replacement	Specify whether to replace built-in data type names with user-defined data type names in generated code.
ReplacementTypes (ERT) string - ''	Code Generation > Data Type Replacement > Data type names	Specify names to use for built-in data types in generated code.

Command-Line Information: Code Generation Pane: Memory Sections Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
MemSecPackage (ERT) string - --- None ---, Simulink, mpt	Code Generation > Memory Sections > Package	Specify the package that contains the memory sections that you want to apply.
MemSecFuncInitTerm (ERT) string - Default , MemConst, MemVolatile, MemConstVolatile	Code Generation > Memory Sections > Initialize/Terminate	Apply memory sections to: <ul style="list-style-type: none"> • Initialize/Start functions • Terminate functions
MemSecFuncExecute (ERT) string - Default , MemConst, MemVolatile, MemConstVolatile	Code Generation > Memory Sections > Execution	Apply memory sections to: <ul style="list-style-type: none"> • Step functions • Run-time initialization functions • Derivative functions • Enable functions • Disable functions
MemSecFuncSharedUtil (ERT) string - Default , MemConst, MemVolatile, MemConstVolatile	Code Generation > Memory Sections > Shared utility	Apply memory sections to shared utility functions.
MemSecDataConstants (ERT)	Code Generation > Memory Sections > Constants	Apply memory sections to: <ul style="list-style-type: none"> • Constant parameters

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
string - Default , MemConst, MemVolatile, MemConstVolatile		<ul style="list-style-type: none"> • Constant block I/O • Zero representation
MemSecDataIO (ERT) string - Default , MemConst, MemVolatile, MemConstVolatile	Code Generation > Memory Sections > Inputs/Outputs	Apply memory sections to: <ul style="list-style-type: none"> • Root inputs • Root outputs
MemSecDataInternal (ERT) string - Default , MemConst, MemVolatile, MemConstVolatile	Code Generation > Memory Sections > Internal data	Apply memory sections to: <ul style="list-style-type: none"> • Block I/O • DWork vectors • Run-time model • Zero-crossings
MemSecDataParameters (ERT) string - Default , MemConst, MemVolatile, MemConstVolatile	Code Generation > Memory Sections > Parameters	Apply memory sections to: <ul style="list-style-type: none"> • Parameters

Command-Line Information: Not in GUI

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CommentStyle (ERT) string - Auto , Multi-line, Single-line	Not available	Specify the comment style for generated C or C++ code: <ul style="list-style-type: none"> • Auto (multi-line for C, single-line for C++) • Multi-line ((/*...*/)) • Single-line (//...)
CPPClassGenCompliant (ERT) string - off, on	Not available	Set in SelectCallback for a target to indicate whether the target supports the ability to generate and configure C++ class interfaces to model code. Default is off for custom and non-ERT

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		targets and on for ERT (<code>ert.tlc</code>) targets. (For more information, see “Support C++ Class Interface Control”.)
ModelStepFunctionPrototypeControlCompliant (ERT) string - off, on	Not available	Set in SelectCallback for a target to indicate whether the target supports the ability to control the function prototypes of initialize and step functions that are generated for a Simulink model. Default is off for non-ERT targets and on for ERT targets. (For more information, see “Support C Function Prototype Control”.)

Model Advisor Checks

Embedded Coder Checks

In this section...

“Embedded Coder Checks Overview” on page 4-2

“Check for blocks not recommended for C/C++ production code deployment” on page 4-2

“Identify lookup table blocks that generate expensive out-of-range checking code” on page 4-3

“Check output types of logic blocks” on page 4-5

“Check the hardware implementation” on page 4-6

“Identify questionable software environment specifications” on page 4-7

“Identify questionable code instrumentation (data I/O)” on page 4-9

“Check for blocks not recommended for MISRA-C:2004 compliance” on page 4-10

“Check configuration parameters for MISRA-C:2004 compliance” on page 4-11

“Identify questionable subsystem settings” on page 4-13

“Identify blocks that generate expensive fixed-point and saturation code” on page 4-13

“Identify questionable fixed-point operations” on page 4-16

“Identify blocks that generate expensive rounding code” on page 4-19

Embedded Coder Checks Overview

Use Embedded Coder Model Advisor checks to configure your model for code generation.

See Also

- “Run Model Checks”
- “Simulink Checks”
- “Simulink Coder Checks”

Check for blocks not recommended for C/C++ production code deployment

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation as identified in the Simulink Block Support tables for Simulink Coder and Embedded Coder. If you are using blocks with support notes for code generation, review the information and follow the given advice.

Available with Simulink Verification and Validation™ and Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- “Supported Products and Block Usage”
- “What Is a Model Advisor Exclusion?”

Identify lookup table blocks that generate expensive out-of-range checking code

Identify lookup table blocks that generate code to protect against out-of-range inputs for breakpoint or index values.

Description

This check verifies that the following blocks do not generate code to protect against inputs that fall outside the range of valid breakpoint values:

- 1-D Lookup Table

- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

This check also verifies that Interpolation Using Prelookup blocks do not generate code to protect against inputs that fall outside the range of valid index values.

Following the recommended actions increases both execution and ROM efficiency of the generated code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The lookup table block generates out-of-range checking code.	Change the setting on the block dialog box so that out-of-range checking code is not generated. <ul style="list-style-type: none"> • For the 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table, and Prelookup blocks, select the check box for Remove protection against out-of-range input in generated code. • For the Interpolation Using Prelookup block, select the check box for Remove protection against out-of-range index in generated code.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

Action Results

Clicking **Modify** prevents lookup table blocks from generating out-of-range checking code, which makes the generated code more efficient.

See Also

- “n-D Lookup Table” block in the Simulink documentation
- “Prelookup” block in the Simulink documentation
- “Interpolation Using Prelookup” block in the Simulink documentation
- “Optimize Generated Code for Lookup Table Blocks” in the Simulink documentation
- “What Is a Model Advisor Exclusion?”

Check output types of logic blocks

Identify logic blocks that do not use `boolean` for the output data type.

Description

This check verifies that the output data type of the following blocks is `boolean`:

- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

Using output data type `boolean` increases execution efficiency of the generated code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The output data type of a logic block is not boolean.	In the block dialog box, set Output data type to boolean .

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- “What Is a Model Advisor Exclusion?”

Action Results

Clicking **Modify** forces logic blocks to use `boolean` as the output data type. If a logic block uses `uint8` for the output type, clicking **Modify** changes the output type to `boolean`.

Check the hardware implementation

Identify inconsistent or underspecified hardware implementation settings

Description

The Simulink and Simulink Coder software require two sets of target specifications. The first set describes the final intended production target. The second set describes the currently selected target. If the configurations do not match, the code generator creates extra code to emulate the behavior of the production target. Inconsistencies or underspecification of hardware attributes can lead to inefficient or incorrect code generation for the target hardware.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Device type is set to Unspecified (assume 32-bit Generic).	In the Configuration Parameters dialog box, on the “ Hardware Implementation ” pane, set Device type to the target hardware.
Hardware implementation parameters are not set to recommended values.	In the Configuration Parameters dialog box, on the “ Hardware Implementation ” pane, specify the following parameters: <ul style="list-style-type: none"> • Byte ordering • Signed integer division rounding
Hardware implementation Production hardware settings do not match Test hardware settings.	In the Configuration Parameters dialog box, on the “ Hardware Implementation ” pane, consider selecting the “ Test hardware is the same as production hardware ” check box, or modify the settings to match.

See Also

Making GRT-Based Targets ERT-Compatible

Identify questionable software environment specifications

Identify questionable software environment settings.

Description

- Support for some software environment settings can lead to inefficient code generation and nonoptimal results.
- Industry standards for C, such as ISO and MISRA, require identifiers to be unique within the first 31 characters.
- Stateflow charts with weak Simulink I/O data types lead to inefficient code.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
The maximum identifier length does not conform with industry standards for C.	In the Configuration Parameters dialog box, on the Code Generation > Symbols pane, set the “ Maximum identifier length ” parameter to 31 characters.
In the Configuration Parameters dialog box, the parameters on the Code Generation > Interface pane are not set to recommended values.	In the Configuration Parameters dialog box, on the “ Code Generation > Interface ” pane, clear the following parameters: <ul style="list-style-type: none"> • Support: continuous time • Support: non-finite numbers • Support: non-inlined S-functions
In the Configuration Parameters dialog box, the parameters on the Code Generation > Symbols pane are not set to recommended values.	In the Configuration Parameters dialog box, on the Code Generation > Symbols pane, set the “ Generate scalar inlined parameters as ” parameter to Literals .
In the Configuration Parameters dialog box, on the Code Generation > Interface pane, Support: variable-size signals is selected. This might lead to inefficient code.	If you do not intend to support variable-sized signals, clear the Code Generation > Interface > “Support: variable-size signals” check box in the Configuration Parameters dialog box.
The model contains Stateflow charts with weak Simulink I/O data type specifications.	Select the Stateflow chart property “ Use Strong Data Typing with Simulink I/O ”. You might need to adjust the data types in your model after selecting the property.

Limitations

A Stateflow license is required when using Stateflow charts.

See Also

“Strong Data Typing with Simulink I/O”

Identify questionable code instrumentation (data I/O)

Identify questionable code instrumentation.

Description

- Instrumentation of the generated code can cause nonoptimal results.
- Test points require global memory and are not optimal for production code generation.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Interface parameters are not set to recommended values.	In the Configuration Parameters dialog box, on the Code Generation > Interface pane, set the parameters to the recommended values.
Blocks generate assertion code.	In the Configuration Parameters dialog box, on the Diagnostics > Data Validity pane, set the “ Model Verification block enabling ” parameter to Disable All on a block-by-block basis or globally.
Block output signals have one or more test points and, if you have an Embedded Coder license, the Ignore test point signals check box is cleared in the Code Generation pane of the Configuration Parameters dialog box.	Remove test points from the specified block output signals. For each signal, in the Signal Properties dialog box, clear the Test point check box. Alternatively, if the model is using an ERT-based system target file, select the Ignore test point signals check box on the Code Generation pane in the Configuration Parameters dialog box to ignore test points during code generation.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- “What Is a Model Advisor Exclusion?”

Check for blocks not recommended for MISRA-C:2004 compliance

Identify blocks that are not supported or recommended for MISRA-C:2004 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA-C:2004 compliant code for embedded applications.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Blocks that are not supported or recommended for MISRA-C:2004 compliant code generation were found in the model or subsystem. For a list of blocks, see “hisl_0020: Blocks not recommended for MISRA-C:2004 compliance”.	Consider replacing the specified blocks.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Verification and Validation license.

See Also

- “hisl_0020: Blocks not recommended for MISRA-C:2004 compliance”
- “MISRA C Guidelines” in the Embedded Coder documentation.
- “MISRA-C:2004 Compliance Considerations”
- “What Is a Model Advisor Exclusion?”

Check configuration parameters for MISRA-C:2004 compliance

Identify configuration parameters that might impact MISRA-C:2004 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA-C:2004 compliant code for embedded applications.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Model Verification block enabling is set to Use local settings or Enable All.	In the Configuration Parameters dialog box, on the Diagnostics > Data Validity pane, set Model Verification block enabling to Disable All.
System target file is set to a GRT-based target.	In the Configuration Parameters dialog box, on the Code Generation > General pane, set System target file to an ERT-based target.
Code Generation > Interface parameters are not set to the recommended values.	In the Configuration Parameters dialog box, on the Code Generation > Interface pane: <ul style="list-style-type: none"> • Clear Support: non-finite numbers • Clear Support: continuous time (ERT-based target only) • Clear Support: non-inlined S-functions (ERT-based target only) • Clear MAT-file logging • Set Standard math library to C89/ C90 (ANSI) • Set Code replacement library to None

Condition	Recommended Action
Parenthesis level is not set to Maximum (Specify precedence with parentheses).	In the Configuration Parameters dialog box, on the Code Generation > Code Style pane, set Parenthesis level to Maximum (Specify precedence with parentheses).
Maximum identifier length is not set to 31 .	In the Configuration Parameters dialog box, on the Code Generation > Symbols pane, set Maximum identifier length to 31 .
Casting Modes is not set to Standards Compliant .	In the Configuration Parameters dialog box, on the Code Generation > Code Style pane, set Casting Modes to Standards Compliant .
<code>GenerateSharedConstants</code> is set to on .	Set <code>GenerateSharedConstants</code> to off .
System-generated identifiers is set to Classic .	In the Configuration Parameters dialog box, on the Code Generation > Symbols pane, set System-generated identifiers to Shortened .
If Pack Boolean data into bitfields is selected and Bitfield declarator type specifier is set to <code>uchar_T</code> .	In the Configuration Parameters dialog box, on the Optimization > Signals and Parameters pane, if Pack Boolean data into bitfields is selected, set Bitfield declarator type specifier to <code>uint_T</code> .

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Capabilities and Limitations

This check does not review referenced models.

See Also

- “`hisl_0060`: Configuration parameters that improve MISRA-C:2004 compliance”
- “`hisl_0313`: Selection of bitfield data types to improve MISRA-C:2004 compliance”
- “MISRA C Guidelines” in the Embedded Coder documentation.

- “MISRA-C:2004 Compliance Considerations”

Identify questionable subsystem settings

Identify questionable subsystem block settings.

Description

Subsystem blocks implemented as void/void functions in the generated code use global memory to store the subsystem I/O.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Subsystem blocks have the Subsystem Parameters > Function packaging option set to Nonreusable function .	Set the Subsystem Parameters > Function packaging parameter to Auto .

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- Subsystem block
- “What Is a Model Advisor Exclusion?”

Identify blocks that generate expensive fixed-point and saturation code

Identify fixed-point operations that can lead to nonoptimal results.

Description

Certain block settings can lead to expensive fixed-point and saturation code.

Results and Recommended Actions

Conditions	Recommended Action
Blocks generate expensive saturation code.	Check whether your application requires setting Function Block Parameters > Signal Attributes > Saturate on integer overflow . Otherwise, clear the Saturate on integer overflow parameter for the most efficient implementation of the block in the generated code.
Product blocks are multiplying signals with mismatched slope adjustment factors. The net slope computation uses multiplication followed by shifts, which is inefficient for some target hardware.	Set the Optimization > Use division for fixed-point net slope computation parameter to On , or Use division for reciprocals of integers only if the net slope can be approximated by a fraction and division is more efficient than multiplication and shifts on the target hardware. Note: This optimization takes place only if certain simplicity and accuracy conditions are met. For more information, see “Handle Net Slope Computation” in the Fixed-Point Designer documentation.
Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.	Reverse the inputs so the multiply operation occurs first and the division operation occurs second.
Product blocks are configured to do multiple division operations.	Multiply all the denominator terms together, and then do a single division using cascading Product blocks.
Product blocks are configured to do many multiplication or division operations.	Split the operations across several blocks, with each block performing one multiplication or one division operation.
Protection code generated as part of the division operation is redundant.	Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the Optimization > Remove

Conditions	Recommended Action
<p>The data type range of the inputs of Sum blocks exceeds the data type range of the output, which can cause overflow or saturation.</p>	<p>code that protects against division arithmetic exceptions” parameter in the Configuration Parameters dialog box.</p> <p>Change the output and accumulator data types so the range equals or exceeds all input ranges.</p> <p>For example, if the model has two inputs</p> <ul style="list-style-type: none"> • <code>int8</code> (–128 to 127) • <code>uint8</code> (0 to 255) <p>The data type range of the output and accumulator must equal or exceed –128 to 255. A <code>int16</code> (–32768 to 32767) data type meets this condition.</p>
<p>A Sum block has an input with a slope adjustment factor that does not equal the slope adjustment factor of the output.</p>	<p>Change the data types so the inputs, outputs, and accumulator have the same slope adjustment factor.</p>
<p>The net sum of the Sum block input biases does not equal the bias of the output.</p>	<p>Change the bias of the output scaling, making the net bias adjustment zero.</p>
<p>The input and output of the MinMax block have different data types.</p>	<p>Change the data type of the input or output.</p>
<p>The input of the MinMax block has a different slope adjustment factor than the output.</p>	<p>Change the scaling of the input or the output.</p>
<p>The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output.</p>	<p>Set the Function Block Parameters > Initial condition setting parameter to State (most efficient).</p>
<p>Parameter overflow occurred for the Compare to Zero block. This block uses the input data type to represent zero. The input data type cannot represent zero exactly, so the input value was compared to the closest representable value of zero.</p>	<p>Select an input data type that can represent zero.</p>

Conditions	Recommended Action
Parameter overflow occurred for the following Compare to Constant block. This block uses the input data type to represent its Constant value parameter. The Constant value parameter is outside the range that the input data type can represent. The input signal was compared to the closest representable value of the Constant value parameter.	Choose an input data type that can represent the Constant value parameter or change the Constant value parameter to match the input data type.

Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- “Identify Blocks that Generate Expensive Fixed-Point and Saturation Code”
- “What Is a Model Advisor Exclusion?”

Identify questionable fixed-point operations

Identify fixed-point operations that can lead to nonoptimal results.

Description

Less efficient code can result from blocks that generate cumbersome multiplication and division operations, expensive conversion code, inefficiencies in lookup table blocks, and expensive comparison code.

Results and Recommended Actions

Conditions	Recommended Action
Integer division generated code is large.	In the Configuration Parameters dialog box, on the Hardware Implementation pane, set the “ Signed integer division rounds to ” parameter to the recommended value.

Conditions	Recommended Action
Lookup Table vector of input values is not evenly spaced.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_look1_func_approx</code> .
Lookup Table vector of input values is not evenly spaced when quantized, but it is very close to being evenly spaced.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_evenspace_cleanup</code> .
Lookup Table vector of input values is evenly spaced, but the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_look1_func_approx</code> .
For a Prelookup or n-D Lookup Table block, Index search method is Evenly spaced points . Breakpoint data does not have power of 2 spacing.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. Otherwise, in the block parameter dialog box, specify a different Index search method to avoid the computation-intensive division operation.
n-D Lookup Table breakpoint data is not evenly spaced and Index search method is not Evenly spaced points .	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing and then set Index search method to Evenly spaced points .
n-D Lookup Table breakpoint data is evenly spaced and Index search method is Evenly spaced points . But the spacing is not a power of 2.	If breakpoint data is nontunable, adjust the data to have even, power of 2 spacing. See <code>fixpt_look1_func_approx</code> .
n-D Lookup Table breakpoint data is evenly spaced, but the spacing is not a power of 2. Also, Index search method is not Evenly spaced points .	Set Index search method to Evenly spaced points . Also, if the data is nontunable, consider an even, power of 2 spacing.
n-D Lookup Table breakpoint data is evenly spaced, and the spacing is a power of 2. But the Index search method is not Evenly spaced points .	Set Index search method to Evenly spaced points .

Conditions	Recommended Action
Blocks require cumbersome multiplication.	Restrict multiplication operations: <ul style="list-style-type: none"> • So the product integer size is not larger than the target integer size. • To the recommended size.
Product blocks are multiplying signals with mismatched slope adjustment factors.	Change the scaling of the output so that its slope adjustment factor is the product of the input slope adjustment factors.
Blocks multiply signals with nonzero bias.	Insert a Data Type Conversion block before and after the block containing the multiplication operation.
The inputs of the Relational Operator block have different data types.	<ul style="list-style-type: none"> • Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type.
The inputs of the Relational Operator block have different slope adjustment factors.	Change the scaling of either input.
The output of the Relational Operator block is constant. This might result in dead code which will be eliminated by Simulink Coder.	Review your model design and either remove the Relational Operator block or replace it with the constant.

Capabilities and Limitations

- A Fixed-Point Designer license is required to generate fixed-point code.
- If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- 1-D Lookup Table
- n-D Lookup Table
- Prelookup
- “Identify Questionable Fixed-Point Operations”

- “What Is a Model Advisor Exclusion?”

Identify blocks that generate expensive rounding code

Check for blocks that generate expensive rounding code.

Description

Generated rounding code is inefficient because of **Integer rounding mode** parameter setting.

Available with Embedded Coder.

Results and Recommended Actions

Condition	Recommended Action
Generated code is inefficient.	Set the Function Block Parameters > Integer rounding mode parameter to the recommended value.

Capabilities and Limitations

If you have a Simulink Verification and Validation license, you can exclude blocks and charts from this check.

See Also

- “Identify Blocks that Generate Expensive Rounding Code”
- “What Is a Model Advisor Exclusion?”

Tools — Alphabetical List

Code Replacement Tool

Create, modify, and validate content of code replacement libraries

Description

The Code Replacement Tool is a graphical interface that you can use to create and manage custom code replacement libraries. You can create, import, manipulate, and validate the code replacement tables in a library. The tool also generates the customization file to register a code replacement library with the code generator. When you open the tool, if you specify a table name, the tool displays only the contents of that table.

Table Information

- Middle pane summarizes table entries.
- Right pane lists table properties.

Field	Description
Name	Name of the code replacement table.
Version	Version number of the table.
Total number of entries	Number of entries in the table.

Entry Summary Information

The middle pane displays summary information for each entry that is in the selected code replacement table.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, <code>cos</code> or <code>RTW_OP_ADD</code>).
Implementation	Name of the implementation function, which can match or differ from Name .

Field	Description
NumIn	Number of input arguments.
InnType	Data type of a conceptual input argument.
OutnType	Data type of a conceptual output argument.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.

Open the Code Replacement Tool

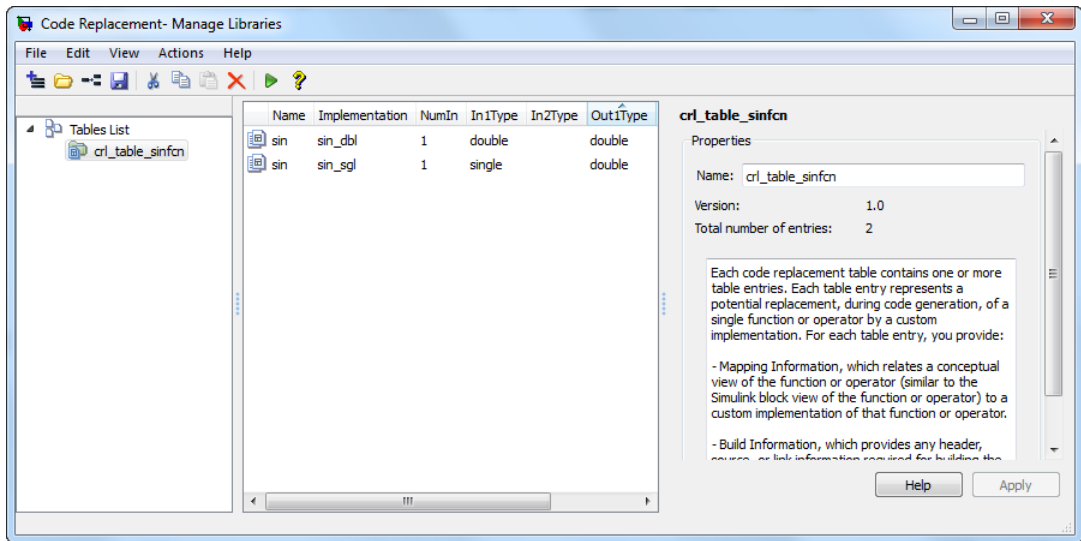
At the command prompt, type `crtool`.

Examples

Open an Existing Table in the Tool

This example shows how to open a code replacement table, `cr1_table_sinfcn`, in the Code Replacement Tool.

```
crtool('cr1_table_sinfcn')
```



- “Quick Start Library Development”
- “Identify Code Replacement Requirements”
- “Prepare for Code Replacement Library Development”
- “Define Code Replacement Mappings”
- “Specify Build Information for Replacement Code”
- “Register Code Replacement Mappings”

Parameters

Mapping Information

The **Mapping Information** tab in the right pane shows conceptual and implementation information for a selected code replacement entry. Available fields vary depending on the entry type, which can be one of:

- Math Operation
- Function
- BLAS Operation

- CBLAS Operation
- Net Slope Fixed Point Operation
- Semaphore Entry (semaphore or mutex)
- Customization Entry (enable inlining or non-finite support for a function)

Except for **Customization Entry**, mapping information specifies a conceptual representation and an implementation (replacement) representation. The conceptual information names the entry and contains match criteria that the code generator uses to find code to replace. Match criteria includes conceptual arguments with corresponding input and output specifications. Match criteria includes other attributes, such as an algorithm, fixed-point saturation, and rounding modes.

The implementation representation specifies C or C++ replacement code, which consists of a function name (for example, 'cos_dbl' or 'u8_add_u8_u8'), arguments with corresponding input and output specifications, and attributes such as:

- Integer saturation mode
- Rounding mode
- Whether an input argument can be an expression
- Whether the function can modify internal or global state

The **Mapping Information** tab also includes a **Validate entry** button you can use to validate new and modified entries.

Build Information

In the right pane, the **Build Information** tab shows information that the code generator uses to generate and build code when using an entry and a match occurs.

- Header file for the replacement function implementation (for example, my_rep_func.h).
- Source file for the replacement function implementation (for example, my_rep_func.c).
- Names and paths of additional header files to include (for example, support_files.h and matlab\customization\mylib\include).
- Names and paths of additional source files to include (for example, support_files.c and matlab\customization\mylib\src).

- Names and paths of link object files to use (for example, `support_files.o` and `matlab\customization\mylib\bin`).
- Link flags (for example, `-MD -Gy`).
- Compile flags (for example, `-Zi -Wall`).
- Whether to copy files from external folders to the build folder before starting the build process.

Programmatic Use

`crtool(table)` opens the Code Replacement Tool and displays the contents of `table`, where `table` is a string that names a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

More About

- “What Is Code Replacement?”
- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

Code Replacement Viewer

Explore content of code replacement libraries

Description

The Code Replacement Viewer displays the content of code replacement libraries. Use the tool to explore and choose a library. If you are developing a custom code replacement library, use the tool to verify table entries.

- Argument order is correct.
- Conceptual argument names match code generator naming conventions.
- Implementation argument names are correct.
- Header or source file specification is not missing.
- I/O types are correct.
- Relative priority of entries is correct (highest priority is 0, and lowest priority is 100).
- Saturation or rounding mode specifications are not missing.

If you specify a library name when you open the viewer, the viewer displays the code replacement tables that the library contains. When you select a code replacement table, the viewer displays function and operator code replacement entries that are in that table.

Abbreviated Entry Information

In the middle pane, the viewer displays entries that are in the selected code replacement table, along with abbreviated information for each entry.

Field	Description
Name	Name or identifier of the function or operator being replaced (for example, <code>COS</code> or <code>RTW_OP_ADD</code>).
Implementation	Name of the implementation function, which can match or differ from Name .
NumIn	Number of input arguments.
In1Type	Data type of the first conceptual input argument.
In2Type	Data type of the second conceptual input argument.
OutType	Data type of the conceptual output argument.

Field	Description
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
UsageCount	Not used.

Detailed Entry Information

In the middle pane, when you select an entry, the viewer displays entry details.

Field	Description
Description	Text description of the table entry (can be empty).
Key	Name or identifier of the function or operator being replaced (for example, <code>COS</code> or <code>RTW_OP_ADD</code>), and the number of conceptual input arguments.
Implementation	Name of the implementation function, and the number of implementation input arguments.
Implementation type	Type of implementation: <code>FCN_IMPL_FUNCT</code> for function or <code>FCN_IMPL_MACRO</code> for macro.
Saturation mode	Saturation mode that the implementation function supports. One of: <code>RTW_SATURATE_ON_OVERFLOW</code> <code>RTW_WRAP_ON_OVERFLOW</code> <code>RTW_SATURATE_UNSPECIFIED</code>
Rounding modes	Rounding modes that the implementation function supports. One or more of: <code>RTW_ROUND_FLOOR</code> <code>RTW_ROUND_CEILING</code> <code>RTW_ROUND_ZERO</code> <code>RTW_ROUND_NEAREST</code> <code>RTW_ROUND_NEAREST_ML</code> <code>RTW_ROUND_SIMPLEST</code>

Field	Description
	RTW_ROUND_CONV RTW_ROUND_UNSPECIFIED
GenCallback file	Not used.
Implementation header	Name of the header file that declares the implementation function.
Implementation source	Name of the implementation source file.
Priority	The entry's match priority, relative to other entries of the same name and to the conceptual argument list within the selected code replacement library. The priority can range from 0 to 100, with 0 being the highest priority. The default is 100. If the library provides two implementations for a function or operator, the implementation with the higher priority shadows the one with the lower priority.
Total Usage Count	Not used.
Entry class	Class from which the current table entry is instantiated.
Conceptual arguments	Name, I/O type (RTW_IO_OUTPUT or RTW_IO_INPUT), and data type for each conceptual argument.
Implementation	Name, I/O type (RTW_IO_OUTPUT or RTW_IO_INPUT), data type, and alignment requirement for each implementation argument.

Fixed-Point Entry Information

When you select an operator entry that specifies net slope fixed-point parameters, the viewer displays fixed-point information.

Field	Description
Net slope adjustment factor F	Slope adjustment factor (F) part of the net slope factor, $F2^E$, for net slope table entries. You use this factor with fixed-point multiplication and division replacement to map a range of slope and bias values to a replacement function.
Net fixed exponent E	Fixed exponent (E) part of the net slope factor, $F2^E$, for net slope table entries. You use this fixed exponent with fixed-point

Field	Description
	multiplication and division replacement to map a range of slope and bias values to a replacement function.
Slopes must be the same	Indicates whether code replacement request processing must check that the slopes on arguments (input and output) are equal. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.
Must have zero net bias	Indicates whether code replacement request processing must check that the net bias on arguments is zero. You use this information with fixed-point addition and subtraction replacement to disregard specific slope and bias values, and map relative slope and bias values to a replacement function.

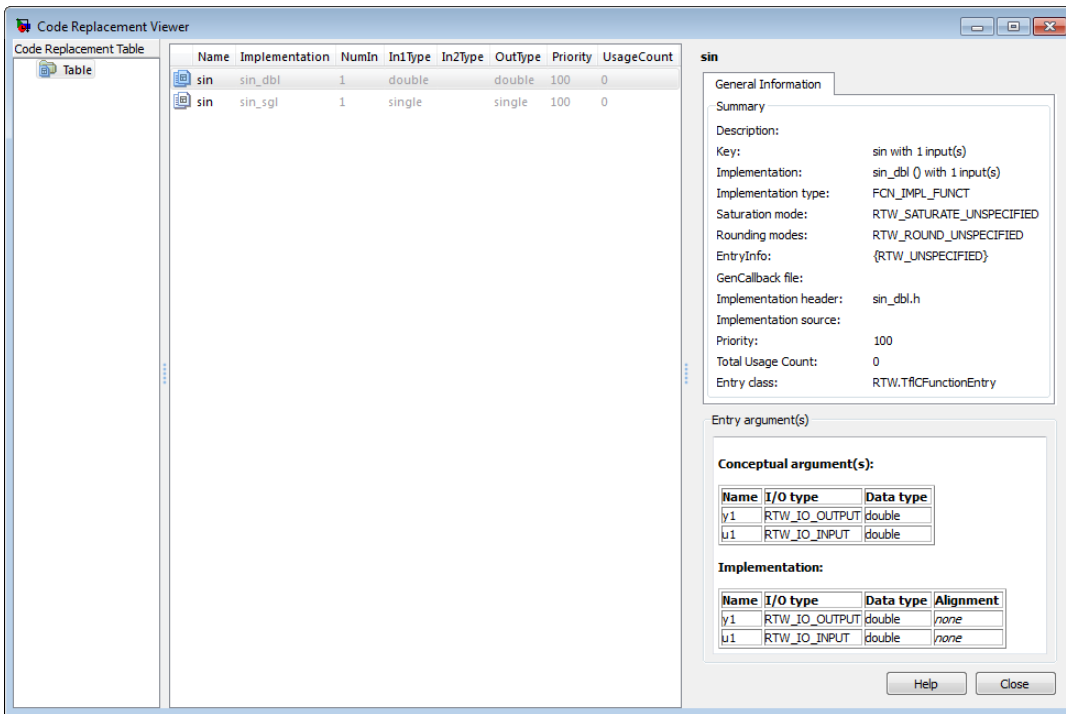
Open the Code Replacement Viewer

Open from the MATLAB command prompt using `RTW.viewTf1`

Examples

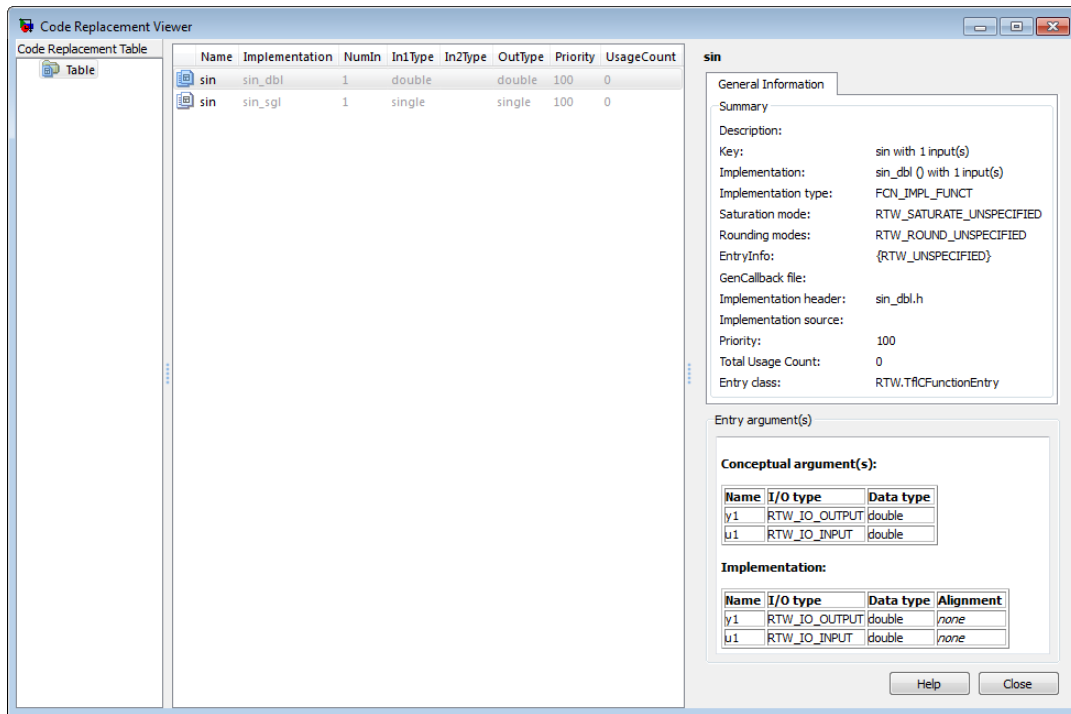
Display Contents of Code Replacement Library

```
RTW.viewTf1('Sin Function Example')
```

Display Contents of Code Replacement Table

RTW.viewTf1(cr1_ttable_sinfcn)



- “Choose a Code Replacement Library”
- “Review and Test Code Replacements”

Programmatic Use

`RTW.viewTf1(library)` opens the Code Replacement Viewer and displays the contents of library, where library is a string that names a registered code replacement library. For example, 'GNU 99 extensions'.

`RTW.viewTf1(table)` opens the Code Replacement Viewer and displays the contents of table, where table is a MATLAB file that defines code replacement tables. The file must be in the current folder or on the MATLAB path.

More About

- “What Is Code Replacement?”

- “What Is Code Replacement Customization?”
- “Code Replacement Libraries”
- “Code Replacement Terminology”

