Embedded Coder™

Reference

R2011b

MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Embedded Coder™ Reference

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)
September 2011	Online only	New for Version 6.1 (Release 2011b)

Contents

1 Function Reference

2 Class Reference

3 Alphabetical List

4 Block Reference

5 Blocks — Alphabetical List

6 Configuration Parameters

1

Function Reference

AUTOSAR

AUTOSAR Component Import

arxml.importer	Construct `arxml.importer` object
createCalibrationComponentObjects (arxml.importer)	Create Simulink calibration objects from AUTOSAR calibration component
createComponentAsModel (arxml.importer)	Create AUTOSAR atomic software component as Simulink model
createComponentAsSubsystem (arxml.importer)	Create AUTOSAR atomic software component as Simulink atomic subsystem
createOperationAsConfigurableSubsystems (arxml.importer)	Create configurable Simulink subsystem library for client-server operation
getApplicationComponentNames (arxml.importer)	Get list of application software component names
getCalibrationComponentNames (arxml.importer)	Get calibration component names
getClientServerInterfaceNames (arxml.importer)	Get list of client-server interfaces
getComponentNames (arxml.importer)	Get application and sensor/actuator software component names
getDependencies (arxml.importer)	Get list of XML dependency files
getFile (arxml.importer)	Return XML file name for `arxml.importer` object

getSensorActuatorComponentNames (arxml.importer)	Get list of sensor/actuator software component names
setDependencies (arxml.importer)	Set XML file dependencies
setFile (arxml.importer)	Set XML file name for `arxml.importer` object

AUTOSAR Configuration

addEventConf (RTW.AutosarInterface)	Add configured AUTOSAR event to model
addIOConf (RTW.AutosarInterface)	Add AUTOSAR I/O configuration to model
attachToModel (RTW.AutosarInterface)	Attach `RTW.AutosarInterface` object to model
getArxmlFilePackaging (RTW.AutosarInterface)	Get AUTOSAR XML packaging format
getComponentName (RTW.AutosarInterface)	Get XML component name
getComponentType (RTW.AutosarInterface)	Get type of software component
getDataTypePackageName (RTW.AutosarInterface)	Get XML data type package name
getDefaultConf (RTW.AutosarInterface)	Get default configuration
getEventType (RTW.AutosarInterface)	Get event type
getExecutionPeriod (RTW.AutosarInterface)	Get runnable execution period
getImplementationName (RTW.AutosarInterface)	Get name of XML implementation
getInitEventName (RTW.AutosarInterface)	Get initial event name

getInitRunnableName (RTW.AutosarInterface)	Get initial runnable name
getInterfacePackageName (RTW.AutosarInterface)	Get XML interface package name
getInternalBehaviorName (RTW.AutosarInterface)	Get name of XML file that specifies software component internal behavior
getIOAutosarPortName (RTW.AutosarInterface)	Get I/O AUTOSAR port name
getIODataAccessMode (RTW.AutosarInterface)	Get I/O data access mode
getIODataElement (RTW.AutosarInterface)	Get I/O data element name
getIOErrorStatusReceiver (RTW.AutosarInterface)	Get name of error status receiver port
getIOInterfaceName (RTW.AutosarInterface)	Get I/O interface name
getIOPortNumber (RTW.AutosarInterface)	Get I/O AUTOSAR port number
getIOServiceInterface (RTW.AutosarInterface)	Get port I/O service interface
getIOServiceName (RTW.AutosarInterface)	Get port I/O service name
getIOServiceOperation (RTW.AutosarInterface)	Get port I/O service operation
getIsServerOperation (RTW.AutosarInterface)	Determine whether server is specified
getPeriodicEventName (RTW.AutosarInterface)	Get periodic event name
getPeriodicRunnableName (RTW.AutosarInterface)	Get periodic runnable name
getServerInterfaceName (RTW.AutosarInterface)	Get name of server interface

getServerOperationPrototype (RTW.AutosarInterface)	Get server operation prototype
getServerPortName (RTW.AutosarInterface)	Get server port name
getServerType (RTW.AutosarInterface)	Determine server type
getTriggerPortName (RTW.AutosarInterface)	Get name of Simulink inport that provides trigger data for DataReceivedEvent
removeEventConf (RTW.AutosarInterface)	Remove AUTOSAR event from model
RTW.AutosarInterface	Construct `RTW.AutosarInterface` object
runValidation (RTW.AutosarInterface)	Validate `RTW.AutosarInterface` object against model
setArxmlFilePackaging (RTW.AutosarInterface)	Set AUTOSAR XML packaging format
setComponentName (RTW.AutosarInterface)	Set XML component name
setComponentType (RTW.AutosarInterface)	Set type of software component
setDataTypePackageName (RTW.AutosarInterface)	Specify XML package name for data type
setEventType (RTW.AutosarInterface)	Set type for event
setExecutionPeriod (RTW.AutosarInterface)	Specify execution period for TimingEvent
setImplementationName (RTW.AutosarInterface)	Set name of XML implementation
setInitEventName (RTW.AutosarInterface)	Set initial event name
setInitRunnableName (RTW.AutosarInterface)	Set initial runnable name

setInterfacePackageName (RTW.AutosarInterface)	Set name of XML interface package
setInternalBehaviorName (RTW.AutosarInterface)	Set name of XML file for software component internal behavior
setIOAutosarPortName (RTW.AutosarInterface)	Set AUTOSAR port name
setIODataAccessMode (RTW.AutosarInterface)	Set I/O data access mode
setIODataElement (RTW.AutosarInterface)	Set I/O data element
setIOErrorStatusReceiver (RTW.AutosarInterface)	Set name of error status receiver port
setIOInterfaceName (RTW.AutosarInterface)	Set I/O interface name
setIOServiceInterface (RTW.AutosarInterface)	Set port I/O service interface
setIOServiceName (RTW.AutosarInterface)	Set port I/O service name
setIOServiceOperation (RTW.AutosarInterface)	Set port I/O service operation
setIsServerOperation (RTW.AutosarInterface)	Indicate that server is specified
setPeriodicEventName (RTW.AutosarInterface)	Set periodic event name
setPeriodicRunnableName (RTW.AutosarInterface)	Set periodic runnable name
setServerInterfaceName (RTW.AutosarInterface)	Set name of server interface
setServerOperationPrototype (RTW.AutosarInterface)	Specify operation prototype
setServerPortName (RTW.AutosarInterface)	Set server port name

setServerType (RTW.AutosarInterface)	Specify server type
setTriggerPortName (RTW.AutosarInterface)	Specify Simulink inport that provides trigger data for DataReceivedEvent
syncWithModel (RTW.AutosarInterface)	Synchronize configuration with model

C++ Encapsulation Interface Control

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

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

Code Execution Profiling

Summary and Timer

display	Generate message that describes how to open code execution profiling report
getNumSectionProfiles	Get number of profiled code sections
getSectionProfile	Get `rtw.pil.ExecutionProfileSection` object for a profiled code section
getTimerTicksPerSecond	Get number of timer ticks per second
report	Open code execution profiling report
setTimerTicksPerSecond	Set number of timer ticks per second

Section Profile

getMaxTicks	Get maximum number of timer ticks for single invocation of profiled code section
getName	Get name of profiled task
getNumCalls	Total number of calls to profiled code section
getSampleOffset	Get sample offset associated with profiled task
getSamplePeriod	Get sample time associated with profiled task
getSectionNumber	Get number that uniquely identifies profiled code section

getTicks	Get execution times in timer ticks for profiled section of code
getTimes	Get execution times in seconds for profiled section of code
getTotalSelfTicks	Get total number of timer ticks recorded for profiled code section excluding periods in child functions
getTotalTicks	Get total number of timer ticks recorded for profiled code section

Code Generation Objectives Customization

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

Code Generation Verification

activateConfigSet (cgv.CGV)	Activate configuration set of model
addBaseline (cgv.CGV)	Add baseline file for comparison
addConfigSet (cgv.CGV)	Add configuration set
addHeaderReportFcn (cgv.CGV)	Add callback function to execute before executing any input data in object
addInputData (cgv.CGV)	Add input data
addPostExecFcn (cgv.CGV)	Add callback function to execute after each input data file is executes
addPostExecReportFcn (cgv.CGV)	Add callback function to execute after each input data file executes
addPostLoadFiles (cgv.CGV)	Add files required by model
addPreExecFcn (cgv.CGV)	Add callback function to execute before each input data file executes
addPreExecReportFcn (cgv.CGV)	Add callback function to execute before each input data file executes
addTrailerReportFcn (cgv.CGV)	Add callback function to execute after all input data executes
compare (cgv.CGV)	Compare signal data
configModel (cgv.Config)	Determine and change configuration parameter values
copySetup (cgv.CGV)	Create copy of object
createToleranceFile (cgv.CGV)	Create file correlating tolerance information with signal names
displayReport (cgv.Config)	Display results of comparing configuration parameter values
getOutputData (cgv.CGV)	Get output data
getReportData (cgv.Config)	Return results of comparing configuration parameter values

getSavedSignals (cgv.CGV)	Display list of signal names to command line
getStatus (cgv.CGV)	Return execution status
plot (cgv.CGV)	Create plot for signal or multiple signals
run (cgv.CGV)	Execute CGV object
setMode (cgv.CGV)	Specify mode of execution
setOutputDir (cgv.CGV)	Specify folder
setOutputFile (cgv.CGV)	Specify output data file name

Embedded IDEs and Embedded Targets

In this section...

IDE Automation Interface

Analog Devices VisualDSP++

activate	Mark file, project, or build configuration as active
add	Add files to active project in IDE
address	Memory address and page value of symbol in IDE
adivdsp	Create handle object to interact with VisualDSP++ IDE
adivdspsetup	Configure your coder product to interact with VisualDSP++ IDE
build	Build or rebuild current project
cd	Set working folder in IDE
close	Close project in IDE window
dir	Files and folders in current IDE window

display (IDE Object)	Properties of IDE handle
getbuildopt	Generate structure of build tools and options
halt	Halt program execution by processor
info	Information about processor
insert	Insert debug point in file
isrunning	Determine whether processor is executing process
isvisible	Determine whether IDE appears on desktop
listsessions	List existing sessions
load	Load program file onto processor
new	Create project, library, or build configuration in IDE
open	Open project in IDE
profile	Generate real-time execution or stack profiling report
profile	Generate real-time execution or stack profiling report
read	Read data from processor memory
remove	Remove file, project, or breakpoint
reset	Stop program execution and reset processor
run	Execute program loaded on processor
save	Save file
setbuildopt	Set active configuration build options
symbol	Program symbol table from IDE
visible	Set whether IDE window appears while IDE runs

write	Write data to processor memory block
xmakefilesetup	Configure your coder product to generate makefiles

Eclipse IDE

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

profile	Generate real-time execution or stack profiling report
pwd	Working folder used by Eclipse™
read	Read data from processor memory
reload	Reload most recent program file to processor signal processor
remove	Remove file, project, or breakpoint
restart	Reload most recent program file to processor signal processor
run	Execute program loaded on processor
write	Write data to processor memory block
xmakefilesetup	Configure your coder product to generate makefiles

Green Hills MULTI

activate	Mark file, project, or build configuration as active
add	Add files to active project in IDE
address	Memory address and page value of symbol in IDE
build	Build or rebuild current project
cd	Set working folder in IDE
close	Close project in IDE window
connect	Connect IDE to processor
dir	Files and folders in current IDE window
display (IDE Object)	Properties of IDE handle
getbuildopt	Generate structure of build tools and options

ghsmulti	Create handle object to interact with MULTI IDE
ghsmulticonfig	Configure coder product to interact with MULTI IDE
halt	Halt program execution by processor
info	Information about processor
insert	Insert debug point in file
isrunning	Determine whether processor is executing process
list	Information listings from IDE
load	Load program file onto processor
new	Create project, library, or build configuration in IDE
open	Open project in IDE
profile	Generate real-time execution or stack profiling report
profile	Generate real-time execution or stack profiling report
read	Read data from processor memory
regread	Values from processor registers
regwrite	Write data values to registers on processor
reload	Reload most recent program file to processor signal processor
remove	Remove file, project, or breakpoint
reset	Stop program execution and reset processor
restart	Reload most recent program file to processor signal processor
run	Execute program loaded on processor

setbuildopt	Set active configuration build options
symbol	Program symbol table from IDE
write	Write data to processor memory block
xmakefilesetup	Configure your coder product to generate makefiles

Texas Instruments Code Composer Studio 3.3

activate	Mark file, project, or build configuration as active
add	Add files to active project in IDE
address	Memory address and page value of symbol in IDE
animate	Run application on processor to breakpoint
build	Build or rebuild current project
ccsboardinfo	Information about boards and simulators known to IDE
cd	Set working folder in IDE
checkEnvSetup	Configure your coder product to interact with Code Composer Studio
close	Close project in IDE window
configure	Define size and number of RTDX™ channel buffers
dir	Files and folders in current IDE window
disable	Disable RTDX interface, specified channel, or all RTDX channels
display (IDE Object)	Properties of IDE handle

enable	Enable RTDX interface, specified channel, or all RTDX channels
flush	Flush data or messages from specified RTDX channels
getbuildopt	Generate structure of build tools and options
halt	Halt program execution by processor
info	Information about processor
insert	Insert debug point in file
isenabled	Determine whether RTDX link is enabled for communications
isreadable	Determine whether specified memory block can read MATLAB software
isrtdxcapable	Determine whether processor supports RTDX
isrunning	Determine whether processor is executing process
isvisible	Determine whether IDE appears on desktop
iswritable	Determine whether MATLAB can write to specified memory block
list	Information listings from IDE
load	Load program file onto processor
msgcount	Number of messages in read-enabled channel queue
new	Create project, library, or build configuration in IDE
open	Open project in IDE
profile	Generate real-time execution or stack profiling report

profile	Generate real-time execution or stack profiling report
read	Read data from processor memory
readmat	Matrix of data from RTDX channel
readmsg	Read messages from specified RTDX channel
regread	Values from processor registers
regwrite	Write data values to registers on processor
reload	Reload most recent program file to processor signal processor
remove	Remove file, project, or breakpoint
reset	Stop program execution and reset processor
restart	Reload most recent program file to processor signal processor
run	Execute program loaded on processor
save	Save file
setbuildopt	Set active configuration build options
symbol	Program symbol table from IDE
ticcs	Create handle object to interact with Code Composer Studio IDE
visible	Set whether IDE window appears while IDE runs
write	Write data to processor memory block
writemsg	Write messages to specified RTDX channel
xmakefilesetup	Configure your coder product to generate makefiles

Texas Instruments Code Composer Studio 4

checkEnvSetup	Configure your coder product to interact with Code Composer Studio

XMakefile

remoteBuild	Build Simulink-generated code on remote target running Linux
xmakefilesetup	Configure your coder product to generate makefiles

Function Prototype Control

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

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

Model Entry Points

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

Processor-in-the-Loop

Connectivity Configuration

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

Build

rtw.connectivity.MakefileBuilder	Configure makefile-based build process

Timer

rtw.connectivity.Timer	Create and configure timer object for target

Execution Download, Start and Stop

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

Host and Target Communications

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

Host-Side Communications

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

Target-Side Communications

rtw.pil.RtIOStreamApplicationFramework	Configure target-side communications

System Target File Callback Interface

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

Target Function Library Table Creation

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

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

2

Class Reference

AUTOSAR

In this section...

AUTOSAR Component Import

arxml.importer	Control import of AUTOSAR components

AUTOSAR Configuration

RTW.AutosarInterface	Control and validate AUTOSAR configuration

C++ Encapsulation Interface Control

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

Code Generation Objectives Customization

rtw.codegenObjectives.Objective	Customize code generation objectives

Code Generation Verification

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

Function Prototype Control

RTW.ModelSpecificCPrototype	Describe signatures of functions for model

Alphabetical List

Purpose

Mark file, project, or build configuration as active

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices™ VisualDSP++®
- Eclipse IDE
- Green Hills® MULTI®
- Texas Instruments™ Code Composer Studio™ v3

Description

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

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

Input Arguments

`IDE_Obj`

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

`objectname`

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

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

For build configurations, enter `'Debug'`, `'Release'`, or `'Custom'`. Before using the `activate` method on a build configuration, activate the project that contains the build configuration. For more information about configurations, see "Configuration" on page 6-122.

`type`

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

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

IDE support for *type*

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

Examples

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

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

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

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

See Also

build | new | remove

cgv.CGV.activateConfigSet

Purpose 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 any 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

- "Managing Model Configurations"
- "Numerical Equivalence Checking"

Purpose

Add files to active project in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

Before using `add`:

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

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

All Supported File Types and Extensions

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

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

Input Arguments

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

IDE_Obj

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

filename

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

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

`filetype`

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

Examples

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

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

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

See Also

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

cgv.CGV.addBaseline

Purpose

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

How To

- "Numerical Equivalence Checking"

cgv.CGV.addHeaderReportFcn

Purpose

Add callback function to execute before executing any 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. `cgv.CGV.run` calls *`CallbackFcn`* before executing any 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

- "Using Callback Functions"

Purpose

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

Syntax

cgvObj.addPostExecFcn(CallbackFcn)

Description

cgvObj.addPostExecFcn(CallbackFcn) adds a callback function to *cgvObj*. *cgvObj* is a handle to a cgv.CGV object. cgv.CGV.run 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

- "Using Callback Functions"

cgv.CGV.addPostExecReportFcn

Purpose

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. cgv.CGV.run 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

- “Using Callback Functions”

Purpose

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. cgv.CGV.run 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

- “Using Callback Functions”

cgv.CGV.addPreExecReportFcn

Purpose

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. cgv.CGV.run 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

- "Using Callback Functions"

Purpose Add callback function to execute after all 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. `cgv.CGV.run` executes all 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

- "Using Callback Functions"

Purpose

Files and folders in current IDE window

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

See Also

`cd` | `open`

addAdditionalHeaderFile

Purpose

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

Syntax

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

Arguments

`hEntry`

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

`headerFile`

String specifying an additional header file.

Description

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

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 TFL table entry.

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

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

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

See Also

`addAdditionalIncludePath` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

How To

- "Specifying Build Information for Code Replacements"
- "Code Replacement"

addAdditionalIncludePath

Purpose

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

Syntax

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

Arguments

`hEntry`

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

`path`

String specifying the full path to an additional header file.

Description

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

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 TFL table entry.

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

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

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

See Also

`addAdditionalHeaderFile` | `addAdditionalSourceFile` | `addAdditionalSourcePath`

How To

- "Specifying Build Information for Code Replacements"
- "Code Replacement"

addAdditionalLinkObj

Purpose

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

Syntax

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

Arguments

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

`linkObj`
: String specifying an additional link object.

Description

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

Examples

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

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

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

See Also

`addAdditionalLinkObjPath`

How To

- “Specifying Build Information for Code Replacements”
- “Code Replacement”

Purpose

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

Syntax

`addAdditionalLinkObjPath(hEntry, path)`

Arguments

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

`path`
: String specifying the full path to an additional link object.

Description

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

Examples

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

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

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

See Also

`addAdditionalLinkObj`

How To

- "Specifying Build Information for Code Replacements"
- "Code Replacement"

addAdditionalSourceFile

Purpose

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

Syntax

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

Arguments

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

`sourceFile`
: String specifying an additional source file.

Description

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

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 TFL table entry.

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

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

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

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourcePath`

How To

- “Specifying Build Information for Code Replacements”
- “Code Replacement”

addAdditionalSourcePath

Purpose

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

Syntax

`addAdditionalSourcePath(hEntry, path)`

Arguments

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

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

Description

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

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 TFL table entry.

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

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

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

See Also

`addAdditionalHeaderFile` | `addAdditionalIncludePath` | `addAdditionalSourceFile`

How To

- "Specifying Build Information for Code Replacements"
- "Code Replacement"

RTW.ModelSpecificCPrototype.addArgConf

Purpose

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

Syntax

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

Description

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

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

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

Input Arguments

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

Examples

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

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

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

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

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

Alternatives

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

See Also

RTW.ModelSpecificCPrototype.attachToModel

How To

- “Function Prototype Control”

rtw.codegenObjectives.Objective.addCheck

Purpose

Add checks

Syntax

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

Description

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

Input Arguments

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

Examples

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

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

See Also

`Simulink.ModelAdvisor`

How To

- “Creating Custom Objectives”
- “About IDs”

Purpose

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

Syntax

`addConceptualArg(hEntry, arg)`

Arguments

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

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

Description

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

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.TflTable;

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

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

```
op_entry.addConceptualArg( arg );

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

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

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

See Also

`getTflArgFromString`

How To

- “Creating Code Replacement Tables”
- “Code Replacement”

Purpose

Add configuration set

Syntax

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

Description

cgvObj.addConfigSet(configSet) is an optional method that adds the configuration set to the object. *cgvObj* is a handle to a cgv.CGV object. *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 all configuration parameter values in the model with the values from the configuration set that you add. The object applies the configuration set when you call the run method. You can add only one configuration set for each cgv.CGV object.

How To

- "Numerical Equivalence Checking"
- "Managing Model Configurations"

addEntry

Purpose

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

Syntax

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

Arguments

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

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

Description

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

Examples

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

```
hLib = RTW.TflTable;

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

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

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

```
op_entry.addConceptualArg( arg );

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

op_entry.copyConceptualArgsToImplementation();

addEntry(hLib, op_entry);
```

How To

- "Creating Code Replacement Tables"
- "Code Replacement"

RTW.AutosarInterface.addEventConf

Purpose

Add configured AUTOSAR event to model

Syntax

```
autosarInterfaceObj.addEventConf('TimingEvent', EventName,
   ExecutionPeriod);
autosarInterfaceObj.addEventConf('DataReceivedEvent',
   EventName, SimulinkInportName);
```

Description

autosarInterfaceObj.addEventConf('TimingEvent', *EventName*, *ExecutionPeriod*); adds a named TimingEvent with a specific execution period.

autosarInterfaceObj.addEventConf('DataReceivedEvent', *EventName*, *SimulinkInportName*); adds a named DataReceivedEvent that triggers a runnable whenever there is a change in value at the specified Simulink inport.

Each call adds a AUTOSAR RTEEvent to *autosarInterfaceObj*, a model-specific RTW.AutosarInterface object.

Input Arguments

TimingEvent

Periodic event that triggers execution of runnable by AUTOSAR Runtime Environment

EventName

Name of AUTOSAR event, which is used in XML description file

ExecutionPeriod

Execution period for AUTOSAR runnable, for example, 0.001.

DataReceivedEvent

Event that triggers execution of runnable by AUTOSAR Runtime Environment only when the value of a received data element is updated.

SimulinkInportName

Simulink inport that receives trigger data

See Also

`RTW.AutosarInterface.removeEventConf`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

cgv.CGV.addInputData

Purpose

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

- "Numerical Equivalence Checking"

Purpose

Add AUTOSAR I/O configuration to model

Syntax

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

Description

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

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

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

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

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

Input Arguments

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

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

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

How To

- "Preparing a Simulink Model for AUTOSAR Code Generation"

rtw.codegenObjectives.Objective.addParam

Purpose

Add parameters

Syntax

`addParam(obj, paramName, value)`

Description

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

Input Arguments

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

Examples

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

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

See Also

`get_param`

How To

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

Purpose

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

- "Numerical Equivalence Checking"
- "Using Callback Functions"

Purpose

Memory address and page value of symbol in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

Input Arguments

`a`

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

`IDE_Obj`

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

`symbol`

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

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

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

`scope`

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

Output Arguments

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

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

For Code Composer Studio

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

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

For Eclipse

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

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

For MULTI®

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

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

For VisualDSP++®

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

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

Examples

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

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

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

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

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

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

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

MATLAB software returns

```
ddatv =

     3.1416   12.3   0.3679   0.7071
```

See Also

`load` | `read` | `symbol` | `write`

Purpose

Create handle object to interact with VisualDSP++ IDE

Syntax

```
IDE_Obj = adivdsp
IDE_Obj = adivdsp('propname1',propvalue1,'propname2',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 any `adivdsp` object property when you create the object, there are several important properties that you must provide during object construction. These properties must be properly 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`, all 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 demonstrate 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

Purpose

Configure your coder product to interact with VisualDSP++ IDE

Syntax

```
adivdspsetup
```

IDEs

This function supports the following IDEs:

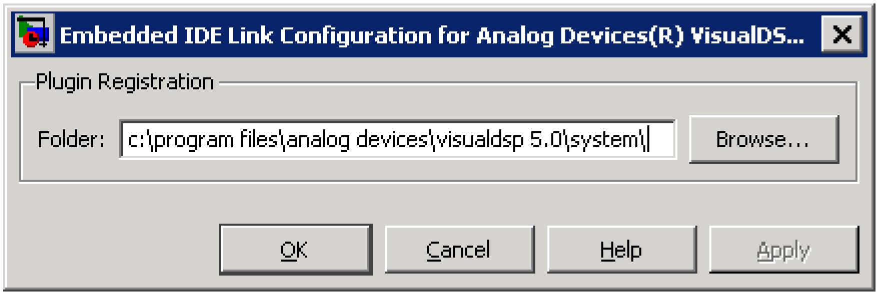
- Analog Devices VisualDSP++

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 any folder 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`

Purpose

Run application on processor to breakpoint

Syntax

IDE_Obj.animate

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

IDE_Obj.animate 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 all 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

arxml.importer

Purpose

Control import of AUTOSAR components

Description

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

Construction

arxml.importer	Construct `arxml.importer` object

Methods

createCalibrationComponentObjects	Create Simulink calibration objects from AUTOSAR calibration component
createComponentAsModel	Create AUTOSAR atomic software component as Simulink model
createComponentAsSubsystem	Create AUTOSAR atomic software component as Simulink atomic subsystem
createOperationAsConfigurableSubsystems	Create configurable Simulink subsystem library for client-server operation
getApplicationComponentNames	Get list of application software component names
getCalibrationComponentNames	Get calibration component names
getClientServerInterfaceNames	Get list of client-server interfaces
getComponentNames	Get application and sensor/actuator software component names

getDependencies	Get list of XML dependency files
getFile	Return XML file name for `arxml.importer` object
getSensorActuatorComponentNames	Get list of sensor/actuator software component names
setDependencies	Set XML file dependencies
setFile	Set XML file name for `arxml.importer` object

Copy Semantics

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

arxml.importer

Purpose Construct `arxml.importer` object

Syntax *importer_obj* = `arxml.importer(`*filename*`)`

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

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

Input Arguments

filename	XML file containing description of atomic software component.

Output Arguments

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

How To

- "Importing an AUTOSAR Software Component"

Purpose

Attach `RTW.AutosarInterface` object to model

Syntax

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

Description

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

Input Arguments

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

How To

- "Modifying and Validating an Existing AUTOSAR Interface"

Purpose

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

Syntax

`attachToModel(obj, modelName)`

Description

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

Input Arguments

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

Alternatives

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

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

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

Syntax

`attachToModel(obj, modelName)`

Description

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

Input Arguments

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

Alternatives

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

How To

- “Function Prototype Control”

Purpose

Build or rebuild current project

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

See Also

`isrunning` | `open`

Purpose

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 `boardnum` 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	`procnum`	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 `procnum`=0, and the first and second processors on the second board are `procnum`=1 and `procnum`=2. You also use this property when you create a link to the IDE.
Processor name	`procname`	Provides the name of the processor. Usually the name is CPU, unless you assign a different name.
Processor type	`proctype`	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 return the table of the information, the method returns a list of board names and numbers. In that list, each board has an 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 Board                          Proc Processor  Processor
Num  Name                            Num  Name       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 Board                         Proc Processor     Processor
Num  Name                           Num  Name          Type
--  ---------------------------------  ---  ------------------
2  C6xxx Simulator (Texas Instrum .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 demonstrate the syntax `boards = ccsboardinfo`, 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 Board                         Proc Processor Processor
Num  Name                           Num  Name      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

Purpose

Set working folder in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

`IDE_Obj``.cd(`*`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 affect 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`

Purpose

Verify numerical equivalence of results

Description

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

`cgv.CGV` and `cgv.Config` use two of the same properties. Before executing a `cgv.CGV` object, it is recommended that you use `cgv.Config` to verify that the model is configured correctly for the mode of execution that you specify. If the top model is set to normal simulation mode, any 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 any order 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` has no effect on 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
addConfigSet	Add configuration set
addHeaderReportFcn	Add callback function to execute before executing any input data in object
addInputData	Add input data
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
addPostLoadFiles	Add files required by model
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 all input data executes
compare	Compare signal data
copySetup	Create copy of object
createToleranceFile	Create file correlating tolerance information with signal names
getOutputData	Get output data

getSavedSignals	Display list of signal names to command line
getStatus	Return execution status
plot	Create plot for signal or multiple signals
run	Execute CGV object
setMode	Specify mode of execution
setOutputDir	Specify folder
setOutputFile	Specify output data file name

Copy Semantics

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

Examples

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

1. Create a `cgv.CGV` object, `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 no longer used for that object. You then use the access methods for verifying the numerical equivalence of the results.

See Also

`cgv.Config`

How To

- "Numerical Equivalence Checking"
- Using Code Generation Verification
- "Verification"

Purpose

Check and modify model configuration parameter values

Description

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

To execute the model successfully in the mode that you specify, you might need to make additional modifications to the configuration parameter values or the model 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. For more information, see `Simulink.ConfigSetRef` in the Simulink documentation.

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

Construction

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

`cfgObj = cgv.Config(model_name, 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 any order as Name1,Value1, ,NameN,ValueN.

Properties

CheckOutports

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

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

ComponentType

Define the SIL or PIL approach

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

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

`Connectivity`

Specify mode of execution

Value	Description
`sim` (default)	Mode of execution is simulation. Recommends changes to a proper subset of the configuration parameters that SIL and all PIL targets require.
`sil`	Mode of execution is SIL. Requires that the system target file is set to `'ert.tlc'` and that you do not use your own external target. Recommends changes to the configuration parameters that SIL targets require.
`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. 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 affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

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

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

See Also

`cgv.CGV`

How To

- “SIL and PIL Simulation”
- “Managing Model Configurations”
- “Programmatic Code Generation Verification”

Purpose

Compare signal data

Syntax

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

Description

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

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

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

Input Arguments

`data_set1, data_set2`

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

`varargin`

Variable number of parameter name and value pairs.

varargin Parameters

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

`Plot`(optional)

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

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

`Signals`(optional)

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

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

The syntax for multiple signal names is:

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

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

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

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

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

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

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

Output Arguments

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

`match_names`

Cell array of matching signal names.

`match_figures`

Array of figure handles for matching signals

`mismatch_names`

Cell array of mismatching signal names

`mismatch_figures`

Array of figure handles for mismatching signals

How To

- "Numerical Equivalence Checking"

cgv.Config.configModel

Purpose Determine and change configuration parameter values

Syntax *`cfgObj`*`.configModel()`

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

How To

- "Verification"
- "Managing Model Configurations"

Purpose

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

Description

Before you use `ticcs` for the first time, use the `checkEnvSetup` function to check for third-party tools and set environment variables. Run `checkEnvSetup` again whenever you configure CCS IDE to interact with a new board or processor, or upgrade any of the related third-party tools.

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

- For *`ide`*, 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
- For *`boardproc`*, enter the name of a supported board or processor. You can get these names from the **Processor** option of the **Custom board for TI CCS** Target Preferences block, located in the `idelinklib_ticcs` block library. For example, enter: `'F2812'`, `'c5509'`, `'c6416dsk'`, `'F2808 eZdsp'`, `'dm6437evm'`.
- For *`action`*, enter the specific action you want this function to perform:
  - `'list'` lists the required third-party tools with their version numbers.
  - `'check'` lists the required third-party tools and the ones on your development system. If any tools are missing, install them. If the version numbers of the tools on your system are not high enough, update the tools.

- 'setup' creates environment variables that point to the installation folders of the third-party tools. If your tools do not meet the requirements, the function advises you. If needed, 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', you can assign the third-party tool information to a variable instead of displaying it on the MATLAB command line. When *action* is 'setup', the statement does not return an output argument.

Examples

To see the required third-party tools and version information for your board, use 'list' as the *action* 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)
```

To compare your versions of the tools with the required versions. Use 'check' as the *action* argument:

```
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    : TFL block replacement
   C64X_IMGLIB_INSTALLDIR="E:\apps\TexasInstruments\C6400\imglib_v1
```

Finally, set the environment variables your coder product requires to use the CCS IDE and generate code for your board. Use `'setup'` as the *`action`* argument, or omit the *`action`* argument:

```
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    : TFL 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) ver
   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:\dvs
  ### Setting environment variable "PSP_EVMDM6437_INSTALLDIR" to "C:\d
  ### Setting environment variable "NDK_INSTALL_DIR" to "C:\dvsdk_1_01
```

Purpose

Close project in IDE window

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

For the *filename* argument:

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

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

Note

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

Examples

To close all open project files:

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

To close the open project, `myProj`:

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

To close the active open project:

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

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

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

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

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

See Also

`add` | `open` | `save`

Purpose

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 that 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=IDE_Obj.rtdx             % 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

Purpose

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 correctly in spite of the error message

Examples

The input argument string`debugconnection` 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`

cgv.CGV.copySetup

Purpose

Create copy of `cgv.CGV` object

Syntax

`cgvObj2 = cgvObj1.copySetup()`

Description

`cgvObj2 = cgvObj1.copySetup()` creates a copy of a `cgv.CGV` object, *`cgvObj1`*. The copied object, `cgvObj2`, has the same configuration as *`cgvObj1`*, but does not copy any 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 `cgv.CGV.setMode`.
- 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 `cgv.CGV.setOutputDir`. 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

- “Numerical Equivalence Checking”

Purpose

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

Syntax

`copyConceptualArgsToImplementation(`*`hEntry`*`)`

Arguments

hEntry

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

Description

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

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.TflTable;

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

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

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

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

op_entry.copyConceptualArgsToImplementation();

hLib.addEntry( op_entry );
```

How To

- “Creating Code Replacement Tables”
- “Code Replacement”

Purpose

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

Syntax

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

Input Arguments

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

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

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

varargin Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 TFL table entry.

Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

How To

- "Creating Code Replacement Tables"
- "Code Replacement"

createAndAddImplementationArg

Purpose

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

Syntax

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

Input Arguments

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

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

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

varargin Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 TFL table entry.

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

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

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

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

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

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

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

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

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

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

See Also

`createAndSetCImplementationReturn`

How To

- “Creating Code Replacement Tables”
- “Code Replacement”

Purpose

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

Syntax

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

Input Arguments

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

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

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

varargin Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 TFL table.

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

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

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

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

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

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

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

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

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

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

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

See Also

`createAndAddImplementationArg`

How To

- "Creating Code Replacement Tables"
- "Code Replacement"

arxml.importer.createCalibrationComponentObjects

Purpose

Create Simulink calibration objects from AUTOSAR calibration component

Syntax

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

Description

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

Input Arguments

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

Output Arguments

success	True if function is successful. False otherwise.

Examples

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

How To

- “Importing an AUTOSAR Software Component”

Purpose

Create AUTOSAR atomic software component as Simulink model

Syntax

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

Description

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

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

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

Input Arguments

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

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

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

Output Arguments

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

Examples

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

How To

- “Importing an AUTOSAR Software Component”

Purpose

Create AUTOSAR atomic software component as Simulink atomic subsystem

Syntax

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

Description

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

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

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

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

```
Simulink.SubSystem.convertToModelReference
```

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

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

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

Input Arguments

`ComponentName`

Absolute short name path of the atomic software component .

`PropertyN, ValueN`

Optional property/value pairs. You can specify values for the following properties:

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

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

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

Output Arguments

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

Examples

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

How To

- "Importing an AUTOSAR Software Component"

arxml.importer.createOperationAsConfigurableSubsystems

Purpose

Create configurable Simulink subsystem library for client-server operation

Syntax

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

Description

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

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

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

Input Arguments

`interfaceName` — Absolute short name path of the client-server interface.

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

`'CreateSimulinkObject'`
: `true` (default) or `false`. If `true`, then the function creates the `Simulink.AliasType` and `Simulink.NumericType` corresponding

to the AUTOSAR data types in the XML file.

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

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

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

Output Arguments

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

Examples

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

See Also

`arxml.importer.getClientServerInterfaceNames`

How To

- “AUTOSAR Communication”
- “Importing an AUTOSAR Software Component”
- “Configuring Client-Server Communication”

cgv.CGV.createToleranceFile

Purpose

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, `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 in the `dataset`. Use `cgv.CGV.getSavedSignals` to view the list of available signal names in the `dataset`. `signal_list` can contain an individual signal or multiple signals. The syntax for an individual signal name is:

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

The syntax for multiple signal names is:

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

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

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

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

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

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

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

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

`tolerance_list`

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

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

How To

- "Numerical Equivalence Checking"

Purpose

Disable RTDX interface, specified channel, or all 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 all 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(IDE_Obj.rtdx,'all') % Disable all open RTDX channels.
```

```
disable(IDE_Obj.rtdx)      % Disable RTDX interface.
```

See Also

close | enable | open

Purpose Properties of IDE handle

Syntax *IDE_Obj*.display()

IDEs This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

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

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

```
IDE_Obj.display

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

See Also get

display

Purpose

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 generated by a SIL or PIL simulation.

See Also

`report`

How To

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"

Purpose Display results of comparing configuration parameter values

Syntax `cfgObj.displayReport()`

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

How To
- “Verification”

eclipseide

Purpose

Create handle object to interact with Eclipse IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

Before using `eclipseide` for the first time:

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

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

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

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

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

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

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

Examples

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

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

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

See Also

`eclipseidesetup`

eclipseidesetup

Purpose

Configure your coder product to interact with Eclipse IDE

Syntax

```
eclipseidesetup
```

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

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

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

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

When to use `eclipseidesetup`:

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

See Also

`eclipseide`

enable

Purpose

Enable RTDX interface, specified channel, or all 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 all 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(IDE_Obj.rtdx) % Enable the RTDX interface.
open(IDE_Obj.rtdx,'inputchannel','w') % Open a channel for sending
                                  % data to the processor.
enable(IDE_Obj.rtdx,'inputchannel') % Enable the channel so you can use
                                % it.
```

See Also

disable | open

Purpose

Enable C++ support for function entry in TFL table

Syntax

`enableCPP(`*`hEntry`*`)`

Arguments

`hEntry`

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

Description

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

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

Examples

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

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

See Also

`registerCPPFunctionEntry` | `setNameSpace`

How To

- “Example: Mapping Math Functions to Target-Specific Implementations”
- “Creating Code Replacement Tables”
- “Code Replacement”

Purpose

Exclude checks

Syntax

`excludeCheck(obj, checkID)`

Description

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

Input Arguments

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

Examples

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

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

See Also

`Simulink.ModelAdvisor`

How To

- “Creating Custom Objectives”
- “About IDs”

flush

Purpose

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 all 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 all 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 all data messages from all RTDX channel queues.

When you use `flush` with a write-configured RTDX channel, your coder product sends all 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 demonstrate `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 = IDE_Obj.rtdx;
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`

arxml.importer.getApplicationComponentNames

Purpose

Get list of application software component names

Syntax

```
applicationSoftwareComponentNames =
importerObj.getApplication
   ComponentNames
```

Description

applicationSoftwareComponentNames = *importerObj*.getApplicationComponentNames returns the names of application software component names found in the XML files associated with *importerObj*, an arxml.importer object.

Output Arguments

applicationSoftwareComponentNames

Cell array of strings. Each element is absolute short-name path of corresponding application software component:

'/*root_package_name*[/*sub_package_name*]/*component_short_name*'

See Also

arxml.importer.getSensorActuatorComponentNames | arxml.importer.getComponentNames

How To

- "Importing an AUTOSAR Software Component"

Purpose

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

Syntax

`category = getArgCategory(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"

- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

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

Syntax

`category = getArgCategory(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Function Prototype Control"

RTW.ModelCPPArgsClass.getArgName

Purpose

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

Syntax

`argName = getArgName(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"

- “C++ Encapsulation Interface Control”

RTW.ModelSpecificCPrototype.getArgName

Purpose

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

Syntax

`argName = getArgName(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Function Prototype Control"

Purpose

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

Syntax

`position = getArgPosition(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

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

Syntax

`position = getArgPosition(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Function Prototype Control"

RTW.ModelCPPArgsClass.getArgQualifier

Purpose

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

Syntax

`qualifier = getArgQualifier(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

RTW.ModelSpecificCPrototype.getArgQualifier

Purpose

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

Syntax

`qualifier = getArgQualifier(obj, portName)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Function Prototype Control"

Purpose

Get AUTOSAR XML packaging format

Syntax

arxmlPackaging = *autosarInterfaceObj*.getArxmlFilePackaging

Description

arxmlPackaging = *autosarInterfaceObj*.getArxmlFilePackaging returns the AUTOSAR XML packaging format in *autosarInterfaceObj*, a model-specific RTW.AutosarInterface object.

Output Arguments

arxmlPackaging

Packaging format of AUTOSAR XML, which is one of the following:.

- 'Modular' — XML descriptions in separate files
- 'Single file' — XML descriptions in single file

See Also

RTW.AutosarInterface.setArxmlFilePackaging

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Exporting AUTOSAR Software Component"

getbuildopt

Purpose

Generate structure of build tools and options

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

`bt=`*`IDE_Obj`*`.getbuildopt` 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=`*`IDE_Obj`*`.getbuildopt(`*`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.

Purpose

Get calibration component names

Syntax

calibrationComponentNames = *importerObj*.getCalibrationComponentNames

Description

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

Output Arguments

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

How To

- "Importing an AUTOSAR Software Component"

RTW.ModelCPPClass.getClassName

Purpose

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

Syntax

`clsName = getClassName(obj)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

Purpose

Get list of client-server interfaces

Syntax

```
interfaceNames = importerObj.getClientServerInterfaceNames
```

Description

interfaceNames = *importerObj*.getClientServerInterfaceNames returns the names of client-server interfaces found in the XML files associated with *importerObj*, an arxml.importer object.

Output Arguments

interfaceNames

Cell array of strings. Each element is absolute short-name path of corresponding client-server interface:

```
'/root_package_name[/sub_package_name]/client_server_interface_short_name'
```

See Also

arxml.importer.createOperationAsConfigurableSubsystems

How To

- "AUTOSAR Communication"
- "Importing an AUTOSAR Software Component"
- "Configuring Client-Server Communication"

Purpose

Get XML component name

Syntax

```
componentName = autosarInterfaceObj.getComponentName
```

Description

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

Output Arguments

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

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get application and sensor/actuator software component names

Syntax

`componentNames = importerObj.getComponentNames`

Description

`componentNames` = *`importerObj`*`.getComponentNames` returns the list of application and sensor/actuator software component names in the XML file associated with the `arxml.importer` object, *`importerObj`*.

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

Output Arguments

`componentNames`	Cell array of strings in which each element is the absolute short name path of the corresponding application software component or sensor/actuator software component: `'/root_package_name[/sub_package_name]/component_short_name'`

See Also

`arxml.importer.getSensorActuatorComponentNames` | `arxml.importer.getApplicationComponentNames`

How To

- "Importing an AUTOSAR Software Component"

Purpose

Get type of software component

Syntax

`componentType = autosarInterfaceObj.getComponentType`

Description

`componentType` `=` *`autosarInterfaceObj`*`.getComponentType` returns the type of the software component in *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Output Arguments

`componentType`

Type of software component. Either `'Application'` or `'Sensor Actuator'`.

See Also

`RTW.AutosarInterface.setComponentType`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get XML data type package name

Syntax

```
dataTypePackageName = autosarInterfaceObj.getDataTypePackageNa
   me
```

Description

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

Output Arguments

`dataTypePackageName`

Name of data type package specified by *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.setDataTypePackageName` |

How To

- "Preparing a Simulink Model for AUTOSAR Code Generation"
- "Generating AUTOSAR Code and Description Files"

RTW.AutosarInterface.getDefaultConf

Purpose

Get default configuration

Syntax

`autosarInterfaceObj.getDefaultConf`

Description

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

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

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

How To

- "Generating Code for AUTOSAR Software Components"

Purpose

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

Syntax

`getDefaultConf(obj)`

Description

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

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

Input Arguments

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

Alternatives

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

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”

- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

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

Syntax

`getDefaultConf(obj)`

Description

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

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

Input Arguments

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

Alternatives

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

How To

- “Function Prototype Control”

arxml.importer.getDependencies

Purpose

Get list of XML dependency files

Syntax

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

Description

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

Output Arguments

`Dependencies`	Cell array of strings.

How To

- "Importing an AUTOSAR Software Component"

Purpose

Get event type

Syntax

`EventType = autosarInterfaceObj.getEventType(EventName)`

Description

`EventType = autosarInterfaceObj.getEventType(EventName)` returns the event type of *EventName*

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

Input Arguments

`EventName`

Name of event

Output Arguments

`EventType`

Type of event, for example, `TimingEvent` or `DataReceivedEvent`

See Also

`RTW.AutosarInterface.setEventType` | `RTW.AutosarInterface.addEventConf`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

RTW.AutosarInterface.getExecutionPeriod

Purpose

Get runnable execution period

Syntax

```
EP = autosarInterfaceObj.getExecutionPeriod
EP = autosarInterfaceObj.getExecutionPeriod(EventName)
```

Description

`EP = autosarInterfaceObj.getExecutionPeriod` returns the execution period of the sole TimingEvent in the runnable.

`EP = autosarInterfaceObj.getExecutionPeriod(EventName)` returns the execution period of a named event in the runnable.

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

Input Arguments

`EventName`

Name of TimingEvent

Output Arguments

`EP`

Execution period of runnable

See Also

`RTW.AutosarInterface.addEventConf` | `RTW.AutosarInterface.setExecutionPeriod`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

Purpose Return XML file name for `arxml.importer` object

Syntax

```
filename = importerObj.getFile
```

Description

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

Output Arguments

`filename`	XML file name

How To

- "Importing an AUTOSAR Software Component"

RTW.ModelSpecificCPrototype.getFunctionName

Purpose

Get function name from model-specific C function prototype

Syntax

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

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- “Function Prototype Control”

Purpose

Get name of XML implementation

Syntax

`implementationName = autosarInterfaceObj.getImplementationName`

Description

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

Output Arguments

`implementationName`	Name of XML implementation for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.setImplementationName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

RTW.AutosarInterface.getInitEventName

Purpose

Get initial event name

Syntax

`initEventName = autosarInterfaceObj.getInitEventName`

Description

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

Output Arguments

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

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get initial runnable name

Syntax

```
initRunnableName = autosarInterfaceObj.getInitRunnableName
```

Description

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

Output Arguments

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

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

RTW.AutosarInterface.getInterfacePackageName

Purpose

Get XML interface package name

Syntax

interfacePkgName = *autosarInterfaceObj*.getInterfacePackageName

Description

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

Output Arguments

interfacePkgName	Name of the interface package specified by *autosarInterfaceObj*

See Also

RTW.AutosarInterface.setInterfacePackageName

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get name of XML file that specifies software component internal behavior

Syntax

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

Description

internalBehaviorName = *autosarInterfaceObj*.getInternalBehaviorName gets the name of the XML file that specifies the software component internal behavior for *autosarInterfaceObj*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Output Arguments

internalBehaviorName	Name of XML file that specifies software component internal behavior for *autosarInterfaceObj*

See Also

RTW.AutosarInterface.setInternalBehaviorName

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Exporting AUTOSAR Software Component"

RTW.AutosarInterface.getIOAutosarPortName

Purpose

Get I/O AUTOSAR port name

Syntax

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

Description

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

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

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

Input Arguments

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

Output Arguments

`ioAutosarName`	AUTOSAR port name of *`portName`*

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get I/O data access mode

Syntax

```
dataAccessMode = autosarInterfaceObj.getIODataAccessMode(portN
   ame)
```

Description

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

Input Arguments

portName	Name of inport/outport (string).

Output Arguments

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

How To

- RTW.AutosarInterface.setIODataAccessMode
- "Preparing a Simulink Model for AUTOSAR Code Generation"

RTW.AutosarInterface.getIODataElement

Purpose

Get I/O data element name

Syntax

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

Description

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

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

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

Input Arguments

portName	Name of inport/outport (string).

Output Arguments

ioDataElement	Data element of the given port (string).

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get name of error status receiver port

Syntax

`ESR = autosarInterfaceObj.getIOErrorStatusReceiver(PortName)`

Description

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

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

Input Arguments

PortName	Name of inport/outport (string)

Output Arguments

ESR	Name of receiver port for *PortName*

See Also

`RTW.AutosarInterface.setIOErrorStatusReceiver`

How To

- “Configuring Ports for Basic Software and Error Status Receivers”

RTW.AutosarInterface.getIOInterfaceName

Purpose

Get I/O interface name

Syntax

```
ioInterfaceName = autosarInterfaceObj.getIOInterfaceName(portN
   ame)
```

Description

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

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

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

Input Arguments

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

Output Arguments

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

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose Get I/O AUTOSAR port number

Syntax `IOPortNumber= autosarInterfaceObj.getIOPortNumber(PortName)`

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

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

PortName	Name of the inport/output (string).

Output Arguments

IOPortNumber	Port number of *PortName*.

How To

- "Generating Code for AUTOSAR Software Components"

RTW.AutosarInterface.getIOServiceInterface

Purpose

Get port I/O service interface

Syntax

`SI = autosarInterfaceObj.getIOServiceInterface(PortName)`

Description

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

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

Input Arguments

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

Output Arguments

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

See Also

`RTW.AutosarInterface.setIOServiceInterface`

How To

- "Configuring Ports for Basic Software and Error Status Receivers"

Purpose

Get port I/O service name

Syntax

`SN = autosarInterfaceObj.getIOServiceName(PortName)`

Description

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

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

Input Arguments

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

Output Arguments

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

See Also

`RTW.AutosarInterface.setIOServiceName`

How To

- "Configuring Ports for Basic Software and Error Status Receivers"

RTW.AutosarInterface.getIOServiceOperation

Purpose

Get port I/O service operation

Syntax

`SO = autosarInterfaceObj.getIOServiceOperation(PortName)`

Description

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

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

Input Arguments

PortName	Inport/outport name (string).

Output Arguments

SO	I/O service operation of *PortName*.

See Also

`RTW.AutosarInterface.setIOServiceOperation`

How To

- "Configuring Ports for Basic Software and Error Status Receivers"

Purpose

Determine whether server is specified

Syntax

`isServerOperation = autosarInterfaceObj.getIsServerOperation`

Description

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

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

Output Arguments

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

How To

- "Configuring Client-Server Communication"

getMaxTicks

Purpose

Get maximum number of timer ticks for single invocation of profiled code section

Syntax

`MaxTicks = NthSectionProfile.getMaxTicks`

Description

`MaxTicks` = *`NthSectionProfile`*`.getMaxTicks` returns the maximum number of timer ticks recorded in a single invocation of the profiled code section during a SIL or PIL simulation.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

Output Arguments

`MaxTicks`

Maximum number of timer ticks for single invocation of profiled code section

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTimes` | `getTicks` | `getSectionNumber` | `getTotalTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- "Configuring Code Execution Profiling"
- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"

Purpose

Get name of profiled task

Syntax

```
SectionName = NthSectionProfile.getName
```

Description

SectionName = *NthSectionProfile*.getName returns the name that identifies the profiled task.

With periodic tasks, the software generates task identifiers that are *based* on the model name. For example, if there is one task, the name of the profiled code section is *model*_step, and if there are two tasks, the names of the profiled code sections are *model*_step0 and *model*_step1. For code that is generated from exported functions, the software uses the name of the function-call inport to identify the task.

NthSectionProfile is an rtw.pil.ExecutionProfileSection object generated by the rtw.pil.ExecutionProfile method getSectionProfile.

Output Arguments

SectionName

Name that identifies profiled code section

See Also

getNumSectionProfiles | getSectionProfile | getTimerTicksPerSecond | setTimerTicksPerSecond | display | report | getSamplePeriod | getSampleOffset | getTicks | getTimes | getSectionNumber | getMaxTicks | getTotalTicks | getTotalSelfTicks | getNumCalls

How To

- “Configuring Code Execution Profiling”
- “Viewing Code Execution Reports”
- “Analyzing Code Execution Data”

RTW.ModelCPPClass.getNumArgs

Purpose

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

Syntax

`num = getNumArgs(obj)`

Description

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

Input Arguments

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

Output Arguments

`num`	An integer specifying the number of step method arguments.

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

Purpose

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

Syntax

`num = getNumArgs(obj)`

Description

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

Input Arguments

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

Output Arguments

num	An integer specifying the number of function arguments.

Alternatives

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

How To

- "Function Prototype Control"

getNumCalls

Purpose

Total number of calls to profiled code section

Syntax

`TotalNumCalls = NthSectionProfile.getNumCalls`

Description

`TotalNumCalls = NthSectionProfile.getNumCalls` returns the total number of calls to the profiled code section over the entire SIL or PIL simulation.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

Output Arguments

`TotalNumCalls`

Total number of calls

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTimes` | `getTicks` | `getSectionNumber` | `getTotalTicks` | `getTotalSelfTicks`

How To

- "Configuring Code Execution Profiling"
- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"

Purpose

Get number of profiled code sections

Syntax

No_of_Sections = *myExecutionProfile*.getNumSectionProfiles

Description

No_of_Sections = *myExecutionProfile*.getNumSectionProfiles returns the number of code sections for which profiling data is available.

There may be cases where, although code sections are instrumented, profiling data is not available because these code sections are not executed.

myExecutionProfile is a workspace variable generated by a SIL or PIL simulation.

Output Arguments

No_of_Sections

Number of code sections with profiling data

See Also

getSectionProfile | getTimerTicksPerSecond | setTimerTicksPerSecond | display | report | getName | getSamplePeriod | getSampleOffset | getTicks | getTimes | getSectionNumber | getMaxTicks | getTotalTicks | getTotalSelfTicks | getNumCalls

How To

- “Configuring Code Execution Profiling”
- “Viewing Code Execution Reports”
- “Analyzing Code Execution Data”

cgv.CGV.getOutputData

Purpose

Get output data

Syntax

`out = cgvObj.getOutputData(InputIndex)`

Description

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

How To

- "Numerical Equivalence Checking"

Purpose

Get periodic event name

Syntax

periodicEventName = *autosarInterfaceObj*.getPeriodicEventName

Description

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

Output Arguments

periodicEventName	Name of the periodic event specified by *autosarInterfaceObj*

Examples

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

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

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

RTW.AutosarInterface.getPeriodicRunnableName

Purpose

Get periodic runnable name

Syntax

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

Description

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

Output Arguments

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

Examples

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

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

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Get model-specific C function prototype code preview

Syntax

`preview = getPreview(obj, fcnType)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Function Prototype Control"

cgv.Config.getReportData

Purpose

Return results of comparing configuration parameter values

Syntax

`rpt_data = cfgObj.getReportData()`

Description

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

How To

- "Verification"

Purpose

Get sample offset associated with profiled task

Syntax

`SampleOffset = NthSectionProfile.getSampleOffset`

Description

`SampleOffset = NthSectionProfile.getSampleOffset` returns the sample offset if the profiled code section is a task.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

Output Arguments

`SampleOffset`

Sample offset associated with profiled task

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getTicks` | `getTimes` | `getSectionNumber` | `getMaxTicks` | `getTotalTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"
- "Analyzing Code Execution Data"

getSamplePeriod

Purpose

Get sample time associated with profiled task

Syntax

SampleTime = *NthSectionProfile*.getSamplePeriod

Description

SampleTime = *NthSectionProfile*.getSamplePeriod returns the sample time if the profiled code section is a task.

NthSectionProfile is an rtw.pil.ExecutionProfileSection object generated by the rtw.pil.ExecutionProfile method getSectionProfile.

Output Arguments

SampleTime

Sample time associated with profiled task

See Also

getNumSectionProfiles | getSectionProfile | getTimerTicksPerSecond | setTimerTicksPerSecond | display | report | getName | getSampleOffset | getTicks | getTimes | getSectionNumber | getMaxTicks | getTotalTicks | getTotalSelfTicks | getNumCalls

How To

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"
- "Analyzing Code Execution Data"

Purpose

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 all output signal names of all data elements from the input data set, *`simulation_data`*. *`simulation_data`* is the output data stored in the CGV object, *`cgvObj`*, when you execute the model.

Tips

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

How To

- "Numerical Equivalence Checking"

getSectionNumber

Purpose

Get number that uniquely identifies profiled code section

Syntax

`SectionNumber = NthSectionProfile.getSectionNumber`

Description

`SectionNumber` = *`NthSectionProfile`*`.getSectionNumber` returns a number that uniquely identifies the profiled code section, for example, in the code execution profiling report.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

Output Arguments

`SectionNumber`

Number of profiled code section

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTimes` | `getTicks` | `getTotalTicks` | `getMaxTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- “Configuring Code Execution Profiling”
- “Configuring Code Execution Profiling”
- “Viewing Code Execution Reports”

Purpose

Get `rtw.pil.ExecutionProfileSection` object for a profiled code section

Syntax

`NthSectionProfile = myExecutionProfile.getSectionProfile(N)`

Description

`NthSectionProfile = myExecutionProfile.getSectionProfile(N)` returns an `rtw.pil.ExecutionProfileSection` object for the *N*th profiled code section.

`myExecutionProfile` is a workspace variable generated by a SIL or PIL simulation.

Use `rtw.pil.ExecutionProfileSection` methods to extract profiling information from the returned object.

Input Arguments

`N`

Index of code section for which profiling data is required

Output Arguments

`NthSectionProfile`

`rtw.pil.ExecutionProfileSection` object that contains profiling information

See Also

`getNumSectionProfiles` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTicks` | `getTimes` | `getSectionNumber` | `getMaxTicks` | `getTotalTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"
- "Analyzing Code Execution Data"

arxml.importer.getSensorActuatorComponentNames

Purpose

Get list of sensor/actuator software component names

Syntax

```
sensoractuatorSoftwareComponentNames =
importerObj.getSensorAc
   tuatorComponentNames
```

Description

`sensoractuatorSoftwareComponentNames` = *`importerObj`*`.getSensorActuatorComponentNames` returns the names of sensor/actuator software component names found in the XML files associated with *`importerObj`*, an `arxml.importer` object.

Output Arguments

`sensoractuatorSoftwareComponentNames`

Cell array of strings. Each element is absolute short-name path of corresponding sensor/actuator software component:

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

See Also

`arxml.importer.getApplicationComponentNames` | `arxml.importer.getComponentNames`

How To

- "Importing an AUTOSAR Software Component"

Purpose

Get name of server interface

Syntax

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

Description

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

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

Output Arguments

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

How To

- “Configuring Client-Server Communication”

RTW.AutosarInterface.getServerOperationPrototype

Purpose

Get server operation prototype

Syntax

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

Description

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

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Output Arguments

operation_prototype	String with names of prototype and arguments: *operation_name*(*dir1 datatype1 arg1*, *dir2 datatype2 arg2*, ..., *dirN datatypeN argN*, ...) • *operation_name* — Name of the operation • *dirN* — Either IN or OUT, which indicates whether data is passed in or out of the function. • *datatypeN* — Data type, which can be an AUTOSAR basic data type or record, Simulink data type, or array. • *argN* — Name of the argument

How To

- "Configuring Client-Server Communication"

Purpose

Get server port name

Syntax

`serverPortName = autosarInterfaceObj.getServerPortName`

Description

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

Output Arguments

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

How To

- "Configuring Client-Server Communication"

RTW.AutosarInterface.getServerType

Purpose

Determine server type

Syntax

`serverType = autosarInterfaceObj.getServerType`

Description

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

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

Output Arguments

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

How To

- "Configuring Client-Server Communication"

Purpose

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 `cgv.CGV.addInputData`.

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.
`pending`	Execution is currently running.
`completed`	Execution ran to completion without any errors and output data is available.
`passed`	Baseline data was provided. Execution ran to completion and comparison to the baseline data returned no differences.

Value	Description
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 (any execution returns 'error')
 status = 'error'
else if (any execution returns 'failed')
 status = 'failed'
else if (any execution returns 'none' or 'pending'
 status = 'none'
```

See Also

`cgv.CGV.addInputData` | `cgv.CGV.run` | `cgv.CGV.addBaseline`

How To

- "Numerical Equivalence Checking"

RTW.ModelCPPClass.getStepMethodName

Purpose

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

Syntax

`fcnName = getStepMethodName(obj)`

Description

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

Input Arguments

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

Output Arguments

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

getTotalSelfTicks

Purpose

Get total number of timer ticks recorded for profiled code section excluding periods in child functions

Syntax

`*TotalSelfTicks* = *NthSectionProfile*.getTotalSelfTicks`

Description

`TotalSelfTicks` = *`NthSectionProfile`*`.getTotalSelfTicks` returns the total number of timer ticks recorded for the profiled code section over the entire SIL or PIL simulation. However, this number excludes the time spent in calls to child functions.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

Output Arguments

`TotalSelfTicks`

Total number of timer ticks for profiled code section

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTimes` | `getTicks` | `getSectionNumber` | `getMaxTicks` | `getTotalTicks` | `getNumCalls`

How To

- “Configuring Code Execution Profiling”
- “Configuring Code Execution Profiling”
- “Viewing Code Execution Reports”

Purpose

Get total number of timer ticks recorded for profiled code section

Syntax

`*TotalTicks* = *NthSectionProfile*.getTotalTicks`

Description

`*TotalTicks* = *NthSectionProfile*.getTotalTicks` returns the total number of timer ticks recorded for the profiled code section over the entire SIL or PIL simulation.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

Output Arguments

`TotalTicks`

Total number of timer ticks for profiled code section

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTimes` | `getTicks` | `getSectionNumber` | `getMaxTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- "Configuring Code Execution Profiling"
- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"

getTflArgFromString

Purpose

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

Syntax

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

Input Arguments

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

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

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

Output Arguments

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

Description

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

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

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 TFL table entry.

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

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

See Also

`addConceptualArg`

How To

- “Creating Code Replacement Tables”
- “Code Replacement”

getTicks

Purpose

Get execution times in timer ticks for profiled section of code

Syntax

```
[ExecutionTimes SelfExecutionTimes] =
NthSectionProfile.getTic
   ks
```

Description

`[`*`ExecutionTimes SelfExecutionTimes`*`] =` *`NthSectionProfile`*`.getTicks` returns two vectors 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, while each element of *`SelfExecutionTimes`* contains the section time excluding the time spent in child functions. 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 an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

If you set the parameter `CodeProfilingSaveOptions` to `'SummaryOnly'` (that is, set **Simulation > Configuration Parameters > Code Generation > SIL and PIL Verification > Save options** to `Summary data only`), then *`NthSectionProfile`*`.getTicks` returns an empty array.

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

See Also

`getNumSectionProfiles | getSectionProfile | getTimerTicksPerSecond | setTimerTicksPerSecond | display | report | getName | getSamplePeriod | getSampleOffset`

| `getTimes` | `getSectionNumber` | `getMaxTicks` | `getTotalTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- "Configuring Code Execution Profiling"
- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"

getTimerTicksPerSecond

Purpose

Get number of timer ticks per second

Syntax

TimerTicksASecond =
myExecutionProfile.getTimerTicksPerSecond

Description

TimerTicksASecond = *myExecutionProfile*.getTimerTicksPerSecond 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 is a workspace variable generated by a SIL or PIL simulation.

Output Arguments

TimerTicksASecond

Number of timer ticks per second

See Also

getNumSectionProfiles | getSectionProfile | setTimerTicksPerSecond | display | report | getName | getSamplePeriod | getSampleOffset | getTicks | getTimes | getSectionNumber | getMaxTicks | getTotalTicks | getTotalSelfTicks | getNumCalls

How To

- “Configuring Code Execution Profiling”
- “Viewing Code Execution Reports”
- “Analyzing Code Execution Data”

Purpose

Get execution times in seconds for profiled section of code

Syntax

```
ExecutionTimes = NthSectionProfile.getTimes
```

Description

`ExecutionTimes` = *`NthSectionProfile`*`.getTimes` returns a vector of execution times, measured in seconds, for the profiled section of code. Each element of the array contains the elapsed time (in seconds) for the profiled section.

The software generates array elements from the timer tick readings, by dividing the readings by the number of timer ticks per second. If you do not specify the number of timer ticks per second, then the method returns an empty array.

`NthSectionProfile` is an `rtw.pil.ExecutionProfileSection` object generated by the `rtw.pil.ExecutionProfile` method `getSectionProfile`.

If you set the parameter `CodeProfilingSaveOptions` to `'SummaryOnly'` (that is, set **Simulation > Configuration Parameters > Code Generation > SIL and PIL Verification > Save options** to `Summary data only`), then *`NthSectionProfile`*`.getTicks` returns an empty array.

Output Arguments

`ExecutionTimes`

Vector of execution times, in seconds, for profiled section of code

See Also

`getNumSectionProfiles` | `getSectionProfile` | `getTimerTicksPerSecond` | `setTimerTicksPerSecond` | `display` | `report` | `getName` | `getSamplePeriod` | `getSampleOffset` | `getTicks` | `getSectionNumber` | `getMaxTicks` | `getTotalTicks` | `getTotalSelfTicks` | `getNumCalls`

How To

- “Configuring Code Execution Profiling”
- “Configuring Code Execution Profiling”
- “Viewing Code Execution Reports”

RTW.AutosarInterface.getTriggerPortName

Purpose

Get name of Simulink inport that provides trigger data for DataReceivedEvent

Syntax

```
SimulinkInportName =
autosarInterfaceObj.getTriggerPortName(Ev
   entName)
```

Description

`SimulinkInportName` = *`autosarInterfaceObj`*`.getTriggerPortName(`*`EventName`*`)` returns the name of the inport that provides trigger data for *`EventName`*, a DataReceivedEvent.

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

Input Arguments

`EventName`

Name of DataReceivedEvent

Output Arguments

`SimulinkInportName`

Name of Simulink inport in model that provides trigger data for *`EventName`*

See Also

`RTW.AutosarInterface.addEventConf` | `RTW.AutosarInterface.setTriggerPortName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

Purpose

Create handle object to interact with MULTI IDE

Syntax

```
IDE_Obj = ghsmulti
IDE_Obj=ghsmulti('propertyname1',propertyvalue1,'propertyname2',...
propertyvalue2,'timeout',value)
```

Note The output object name you provide for `ghsmulti` cannot begin with an underscore, such as `_IDE_Obj`.

IDEs

This function supports the following IDEs:

- Green Hills MULTI

Description

`IDE_Obj = ghsmulti` returns object `IDE_Obj` that communicates with a target processor. Before you use this command for the first time, use `ghsmulticonfig` to configure your MULTI software installation to identify the location of your MULTI software, your processor configuration, your debug server, and the host name and port number of the service.

`ghsmulti` creates an interface between MATLAB and Green Hills MULTI.

The first time you use `ghsmulti`, supply the properties and property values shown in following table as input arguments.

Property Name	Default Value	Description
hostname	localhost	Specifies the name of the machine hosting the service. The default host name indicates that the service is on the local PC. Replace `localhost` with the name you entered as the **Host name** when you ran `ghsmulticonfig`.
portnum	4444	Specifies the port to connect to the service on the host machine. Replace `portnum` with the number you entered as the **Port number** when you ran `ghsmulticonfig`.

When you invoke `ghsmulti`, it starts a service on your localhost. If you selected the **Show server status window** option when you ran `ghsmulticonfig`, the service appears in your Microsoft Windows task bar. If you clear **Show server status window**, the service does not appear.

Parameters that you pass as input arguments to `ghsmulti` are interpreted 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).

`IDE_Obj = ghsmulti('hostname','name','portnum','number',...)` returns a `ghsmulti` object `IDE_Obj` that you use to interact with a processor in the IDE from the MATLAB command prompt. If you enter a `hostname` or `portnum` that are not the same as the ones you provided when you configured your MULTI installation, the software returns an error that it could not connect to the specified host and port and does not create the object.

You use the debugging methods with this object to access memory and control the execution of the processor. `ghsmulti` also enables you to create an array of objects for a multiprocessor board, where each object refers to one processor on the board. When IDE_Obj is an array of objects, any method called with IDE_Obj as an input argument is sent sequentially to all processors connected to the ghsmulti object. Green Hills MULTI provides the communication between the IDE and the processor.

After you build the ghsmulti object IDE_Obj, you can review the object property values with get, but you cannot modify the hostname and portnum property values. You can use set to change the value of other properties.

IDE_Obj=ghsmulti('propertyname1',propertyvalue1,'propertyname2',... propertyvalue2,'timeout',value) sets the global time-out value in seconds to value in IDE_Obj. MATLAB waits for the specified time-out period 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.

Examples

This example demonstrates ghsmulti using default values.

```
IDE_Obj = ghsmulti('hostname','localhost','portnum',4444);
```

returns a handle to the default host and port number—localhost and 4444.

```
IDE_Obj = ghsmulti('hostname','localhost','portnum',4444)

MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\ppc\
```

See Also

ghsmulticonfig

ghsmulticonfig

Purpose Configure coder product to interact with MULTI IDE

Syntax `ghsmulticonfig`

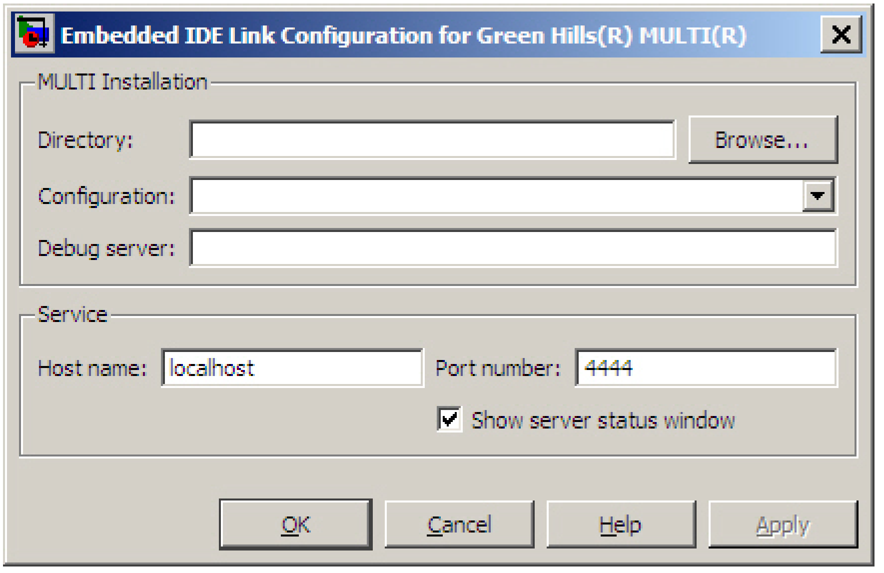
IDEs This function supports the following IDEs:

- Green Hills MULTI

Description `ghsmulticonfig` launches a configuration dialog box to specify information about MULTI.

Note The configuration dialog box is the only place you set the host name and port number configuration.

The dialog box, shown in the following figure, provides controls that specify parameters such as where you installed MULTI and the name of the host machine to use.



Directory

Enter the full path to your Green Hills MULTI executable, `multi.exe`. To search for the executable file, click **Browse**.

If you do not provide or select a correct path to the executable file, the software ignores your entry and returns an error message saying it could not find the executable `multi.exe` in the specified or selected folder.

Configuration

Specifies the primary processor family to use to develop your projects in MULTI. This corresponds to a `.tgt` file you select before you can download and execute code. Select your family file from the list. In many cases, the *`family`*`_standalone.tgt` option is the appropriate choice. For example, if you develop on the Freescale MPC5xx, you could select `ppc_standalone.tgt`. The software stores your selection. You do not need to repeat this setup task unless you change processors.

Debug server

Use this parameter to enter the name of your debug connection. The software uses this connection to specify options about the processor, such as processor to use, board support library, and processor endianness. For more information about the Debug server, refer to your Green Hills MULTI documentation.

For example, if you are using the Freescale MPC5554 simulator, you could enter the string `simppc -cpu=ppc5554 -dec -rom_use_entry`. Valid strings for specifying simulators in **Debug server** appear in the following table.

Processor	Type	Configuration	Debug Server Parameter String
ARM	Simulator	arm_standalone.tgt	simarm -cpu=arm9
MPC5554	Simulator	`ppc_standalone.tgt`	`simppc -cpu=ppc5554 -dec -rom_use_entry`
MPC7400	Simulator	`ppc_standalone.tgt`	`simppc -cpu=7400 -dec`
BlackFin 537	Simulator	`bf_standalone.tgt`	`simbf -cpu=bf537 -fast`
NEC V850	Simulator	`v800_standalone.tgt`	`sim850 -cpu=v850`
NEC V850	NEC Minicube	`v800_standalone.tgt`	`850eserv2 -minicube -noiop -df=C:/ghs/multi505/v850e/df3707.800 -id ffffffffff`
MPC5554	Embedded target Green Hills Probe	`ppc_standalone.tgt`	`mpserv_standard.mbs mpserv -usb`

For information about using hardware in your development work, refer to *Connecting to Your Target* in the MULTI documentation. The string you specify for **Debug server** can be the name of the connection if you have one configured in the Connection Organizer in MULTI.

Host name
Specify the name of the machine that runs the service. Enter `localhost` if the service runs on your PC. `localhost` is the only supported host name.

Port number
Specify the port the service uses to communicate with MULTI. The default port number is 4444. If you change the port value, verify that the port is available for use. If the port you assign is not available, the software returns an error when you try to create a `ghsmulti` object.

Show server status window
Select this option to display the service status in the Microsoft® Windows Task bar. Clearing the option removes the service from the task bar. Best practice is to select this option. Keeping this option selected enables the software to shut down the communication services for Green Hills MULTI completely.

Purpose Halt program execution by processor

Syntax

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

IDEs This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

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

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

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

Examples

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

```
IDE_Obj.isrunning

ans =

     1

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

ans =

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

ans =

     0
```

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

See Also

isrunning | reset | run

Purpose

Information about processor

Syntax

```
adf=IDE_Obj.info
adf = IDE_Obj.info
adf = info(rx)
adf = IDE_Obj.info
adf = info(rx)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

`adf=IDE_Obj.info` 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 = IDE_Obj.info;
```

Using info with `IDE_Obj`, which is associated with 2 processors:

```
oinfo = IDE_Obj.info; % Returns a 1x2 array of infor struct
```

Using info with MULTI IDE

Before using info, open a program in the MULTI IDE debugger. When you use `info` with an IDE handle object for the MULTI IDE, the info method returns the following information.

Structure Element	Data Type	Description
adf.CurBrkPt	String	When the debugger is stopped at a breakpoint, the field reports the index of the breakpoint. Otherwise, this value is-1.
adf.File	String	Name of the current file shown in the debugger source pane.
adf.Line	Integer	Line number of the cursor position in the file in the debugger source pane. If no file is open in the source pane, this value is -1.
adf.MultiDir	String	Full path to your IDE installation the root folder). For example 'C:\ghs5_01'
adf.PID	Double	Process ID from the debug server in the IDE.
adf.Procedure	String	Current procedure in the debugger source pane.
adf.Process	Double	Program number, defined by the IDE, of the current program.
adf.Remote	String	Status of the remote connection, either Connected or Not connected.
adf.Selection	String	The string highlighted in the debugger. If there is no string highlighted, this value is 'null'.

Structure Element	Data Type	Description
adf.State	String	State of the loaded program. The possible reported states appear in the following list: • About to resume • Dying • Just executed • Just forked • No child • Running • Stopped • Zombied For details about the states and their definitions, refer to your IDE debugger documentation.
adf.Target	Double	Unique identifier the indicates the processor family and variant.
adf.TargetOS	Double	Real-time operating system on the processor if one exists. Provides both the major and minor revision information.
adf.TargetSeries	Double	Whether the processor belongs to a series of processors. For details about the processor series, refer to your IDE debugger documentation.

`info` returns valid information when the IDE debugger is connected to processor hardware or a simulator.

Examples

On a PC with a simulator configured in the IDE, `info` returns the following configuration information after stopping a running simulation:

```
adf=info(test_obj1)
```

```
adf =

  CurBrkPt: 0
  File: '...\Compute_Sum_and_Diff_multilink\Compute_Sum_and_Diff_main.c'
  Line: 3
  MultiDir: 'C:\ghs5_01'
  PID: 2380
  Procedure: 'main'
  Process: 0
  Remote: 'Connected'
  Selection: '(null)'
  State: 'Stopped'
  Target: 4325392
  TargetOS: [2x1 double]
  TargetSeries: 3
```

When you create an IDE handle, the response from `info` looks like the following before you load a project.

```
adf=info(test_obj2)

test_obj2 =

        CurBrkPt: []
            File: []
            Line: []
        MultiDir: []
             PID: []
       Procedure: []
         Process: []
          Remote: []
       Selection: []
           State: []
          Target: []
        TargetOS: []
    TargetSeries: []
```

Using info with CCS IDE

`adf = IDE_Obj.info` 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
`adf.procname`	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with *IDE_Obj*.
`adf.isbigendian`	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`.
`adf.family`	Integer	Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors.
`adf.subfamily`	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 `dec2hex` to convert the value in `adf.subfamily` to standard notation. For example `dec2hex(adf.subfamily)` produces `'67'` when the processor is a member of the 67xx processor family.
`adf.timeout`	Integer	Default timeout value MATLAB software uses when transferring data to and from CCS. All functions that use a timeout value have an optional *`timeout`* input argument. When you omit the optional argument, MATLAB software uses 10s as the default value.

`adf = info(rx)` returns info as a cell arraying 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:

```
IDE_Obj.info

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.

```
IDE_Obj.info

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 = *IDE_Obj*.info 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
adf.procname	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with *IDE_Obj*.
adf.proctype	String	String with the type of the DSP processor. The type property is the processor type like "ADSP-21065L" or "ADSP-2181".
adf.revision	String	String with the silicon revision string of the processor.

adf = info(rx) returns info as a cell arraying 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 = IDE_Obj.info

objinfo =

    procname: 'ADSP-BF533'
    proctype: 'ADSP-BF533'
    revision: ''

objinfo.procname

ans =

ADSP-BF533
```

See Also

dec2hex | get | set

Purpose

Insert debug point in file

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

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

See Also

address | run

isenabled

Purpose

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 any 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 reliable 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.

```
IDE_Obj.restart
IDE_Obj.run('run');
IDE_Obj.rtdx.enable('ichan');
IDE_Obj.rtdx.isenabled('ichan')
```

MATLAB software returns 1 indicating that your channel `'ichan'` is enabled for RTDX communications. To determine the mode for the channel, use `IDE_Obj.rtdx`to display the properties of object `IDE_Obj.rtdx`.

See Also

`clear` | `disable` | `enable`

isreadable

Purpose

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

```
IDE_Obj.isreadable(address,'datatype',count)
IDE_Obj.isreadable(address,'datatype')
isreadable(rx,'channel')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`IDE_Obj``.isreadable(`*`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 any 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 all 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 *IDE_Obj*.page
10	Decimal	Address is 10 decimal on the page referred to by *IDE_Obj*.page
[18,1]	Vector	Address location 10 decimal on memory page 1 (*IDE_Obj*.page = 1)

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 *IDE_Obj*.page.

count — A numeric scalar or vector that defines the number of *datatype* values to test for being readable. To assure parallel structure with read, *count* can be a vector to define multidimensional data blocks. This function always 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
`'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 any 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.

`IDE_Obj.isreadable(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 any 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 properly 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 properly.

```
IDE_Obj = ticcs;
rx = IDE_Obj.rtdx;

% Define read and write channels to the processor linked by IDE_Obj.
open(rx,'ichannel','r');s
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`

Purpose

Determine whether processor supports RTDX

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

Syntax

`b=IDE_Obj.isrtdxcapable`

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`b=IDE_Obj.isrtdxcapable` 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=IDE_Obj.isrtdxcapable
b =
   1
```

isrunning

Purpose

Determine whether processor is executing process

Syntax

IDE_Obj.isrunning

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

Examples

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

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

ans =

     1
IDE_Obj.halt
IDE_Obj.isrunning

ans =

     0
```

See Also

halt | load | run

Purpose Determine whether IDE appears on desktop

Syntax *IDE_Obj*.isvisible

IDEs This function supports the following IDEs:

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

Description *IDE_Obj*.isvisible 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:

```
IDE_Obj.isvisible #determine if the ide is visible

ans =

     1
IDE_Obj.visible(0) #make the ide invisible
IDE_Obj.isvisible #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`

Purpose

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

```
IDE_Obj.iswritable(address,'datatype',count)
IDE_Obj.iswritable(address,'datatype')
iswritable(rx,'channel')
```

IDEs

This function supports the following IDEs:

- Texas Instruments Code Composer Studio v3

Description

`IDE_Obj.iswritable(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 any 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 all 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 `IDE_Obj.page`
10	Decimal	Address is 10 decimal on the page referred to by `IDE_Obj.page`
[18,1]	Vector	Address location 10 decimal on memory page 1 (`IDE_Obj.page = 1`)

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 `IDE_Obj.page`.

`count` — A numeric scalar or vector that defines the number of `datatype` values to test for being writable. To assure parallel structure with `write`, `count` can be a vector to define multidimensional data blocks. This function always 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:

```
IDE_Obj.iswritable(31,[10 10 10])
```

`iswritable` writes 1000 values (10*10*10) to the processor. For a two-dimensional matrix defined with `count` as

```
IDE_Obj.iswritable(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

IDE_Obj`.iswritable(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 any 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 properly 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 any 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 properly.

```
IDE_Obj = ticcs;
rx = IDE_Obj.rtdx;

% 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

Purpose Information listings from IDE

Syntax

```
IDE_Obj.infolist = list('type')
IDE_Obj.infolist = list('type',typename)
```

IDEs This function supports the following IDEs:

- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

Using list with MULTI

`infolist = IDE_Obj.list(type)` reads information about your the IDE project and returns it in `infolist`. Different types of information and return formats are possible depending on the input arguments you supply to the `list` function call.

Note `list` does not recognize or return information about variables that you declare in your code but that are not used or initialized.

The *type* argument specifies which information listing to return. To determine the information that `list` returns, use one of the entries in the following table.

type String	Description
project	Return information about the current project in the IDE
variable	Return information about one or more embedded variables
function	Return details about one or more functions in your project

list returns dynamic the IDE information that you can alter. Returned listings represent snapshots of the current the IDE configuration only. Be aware that earlier copies of infolist might contain stale information.

infolist = *IDE_Obj*.list('**project**') returns a vector of structures that contain project information in the format shown in the following table.

infolist Structure Element	Description
infolist(1).name	Project file name (with path).
infolist(1).primary	Configuration file used for the project. For more information, refer to new.
infolist(1).compileroptions	Compiler options string for the project.
infolist(1).srcfiles	Vector of structures that describes project source files. Each structure contains the name and path for each source file—infolist(1).srcfiles.name.
infolist(1).type	Shows the project type, either project or projlib. For more information, refer to new.
infolist(2)....	...
infolist(n)....	...

infolist = *IDE_Obj*.list('**variable**') returns a structure of structures that contains information on all local variables within scope. The list also includes information on all global variables. If a local variable has the same symbol name as a global variable, list returns the local variable information.

infolist = *IDE_Obj*.list('**variable**',varname) returns information about the specified variable varname.

infolist = *IDE_Obj*.list('**variable**',varnamelist) returns information about variables in a list specified by varnamelist. The information returned in each structure follows the format in the following table.

infolist Structure Element	Description
infolist.varname(1).name	Symbol name.
infolist.varname(1).isglobal	Indicates whether symbol is global or local.
infolist.varname(1).location	Information about the location of the symbol.
infolist.varname(1).size	Size per dimension.
infolist.varname(1).uclass	IDE handle class that matches the type of this symbol.
infolist.varname(1).bitsize	Size in bits. More information is added to the structure depending on the symbol type.
infolist.(varname1).type	Data type of symbol.
infolist.varname(2)....	...
infolist.varname(n)....	...

list uses the variable name as the field name to refer to the structure information for the variable.

infolist = *IDE_Obj*.list('**globalvar**') returns a structure that contains information on all global variables.

infolist = *IDE_Obj*.list('**globalvar**',varname) returns a structure that contains information on the specified global variable.

infolist = *IDE_Obj*.list('**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 = IDE_Obj.list('variable',...).

`infolist = IDE_Obj.list('function')` returns a structure that contains information on all functions in the embedded program.

`infolist = IDE_Obj.list('function',functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = IDE_Obj.list('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
`infolist.functionname(1).name`	Function name
`infolist.functionname(1).filename`	Name of file where function is defined
`infolist.functionname(1).address`	Relevant address information such as start address and end address
`infolist.functionname(1).funcvar`	Variables local to the function
`infolist.functionname(1).uclass`	IDE handle class that matches the type of this symbol—**`function`**
`infolist.functionname(1).funcdecl`	Function declaration—where information such as the function return type is contained
`infolist.functionname(1).islibfunc`	Determine if the library is a function

infolist Structure Element	Description
infolist.functionname(1).linepos	Start and end line positions of function
infolist.functionname(1).funcinfo	Miscellaneous information about the function
infolist.functionname(2)...	...
infolist.functionname(n)...	...

To refer to the function structure information, `list` uses the function name as the field name.

`IDE_Obj.infolist = list('type')` returns a structure that contains information on all defined data types in the embedded program. This method includes struct, enum and union data types and excludes typedefs. The name of a defined type is its C struct tag, enum tag or union tag. If the C tag is not defined, it is referred to by the IDE compiler as `'$faken'` where *n* is an assigned number.

`IDE_Obj.infolist = list('type',typename)` returns a structure that contains information on the specified defined data type.

`IDE_Obj.infolist = list('type',typenamelist)` returns a structure that contains information on the specified defined data types in the list. The returned information follows the following format when you specify option `type` as **`type`**.

infolist Structure Element	Description
infolist.typename(1).type	Type name.
infolist.typename(1).size	Size of this type.
infolist.typename(1).uclass	IDE handle class that matches the type of this symbol. Additional information is added depending on the type.

infolist Structure Element	Description
infolist.typename(2)....	...
infolist.typename(n)....	...

For the field name, list uses the type name to refer to the type structure information.

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 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 field name does not change the name of the embedded symbol or type.

Examples

This first example shows list used with a variable, providing information about the variable varname. Notice that the invalid field name _with_underscore gets changed to Q_with_underscore. To make the invalid name valid, list inserts the character Q before the name.

```
varname1 = '_with_underscore'; % Invalid fieldname.
IDE_Obj.list('variable',varname1);
ans =

    Q_with_underscore : [varinfo]
ans. Q_with_underscore
ans=

       name: '_with_underscore'
   isglobal: 0
   location: [1x62 char]
       size: 1
```

```
uclass: 'numeric'
  type: 'int'
bitsize: 16
```

To demonstrate using 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.
IDE_Obj.list('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 the IDE, you see a listing like the following that includes structures containing details about your project.

```
projectinfo=IDE_Obj.list('project')

projectinfo =

          name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'
          type: 'project'
    targettype: 'TMS320C67XX'
      srcfiles: [69x1 struct]
      buildcfg: [3x1 struct]
```

Using list with CCS IDE

`infolist = IDE_Obj.list(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 any 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 = *IDE_Obj*.list('`**`project`**`')`, the method returns a vector of structures that contain project information in the following format.

infolist Structure Element	Description
`infolist(1).name`	Project file name (with path).
`infolist(1).type`	Project type — `project`,`projlib`, or `projext`, refer to `new`.
`infolist(1).processortype`	String description of processor CPU.
`infolist(1).srcfiles`	Vector of structures that describes project source files. Each structure contains the name and path for each source file — `infolist(1).srcfiles.name`.

infolist Structure Element	Description
`infolist(1).buildcfg`	Vector of structures that describe build configurations, each with the following entries: • `infolist(1).buildcfg.name` — the build configuration name. • `infolist(1).buildcfg.outpath` — the default folder for storing the build output.
`infolist(2)....`	...
`infolist(n)....`	...

`infolist = `*`IDE_Obj`*`.list('`**`variable`**`')` returns a structure of structures that contains information on all local variables within scope. The list also includes information on all 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 = `*`IDE_Obj`*`.list('`**`variable`**`',varname)` returns information about the specified variable `varname`.

`infolist = `*`IDE_Obj`*`.list('`**`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
`infolist.varname(1).name`	Symbol name.
`infolist.varname(1).isglobal`	Indicates whether symbol is global or local.
`infolist.varname(1).location`	Information about the location of the symbol.
`infolist.varname(1).size`	Size per dimension.

infolist Structure Element	Description
`infolist.varname(1).uclass`	`ticcs` object class that matches the type of this symbol.
`infolist.varname(1).bitsize`	Size in bits. More information is added to the structure depending on the symbol type.
`infolist.varname(2)....`	...
`infolist.varname(n)....`	...

`list` uses the variable name as the field name to refer to the structure information for the variable.

`infolist = IDE_Obj.list('globalvar')` returns a structure that contains information on all global variables.

`infolist = IDE_Obj.list('globalvar',varname)` returns a structure that contains information on the specified global variable.

`infolist = IDE_Obj.list('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 = IDE_Obj.list('variable',...)`.

`infolist = IDE_Obj.list('function')` returns a structure that contains information on all functions in the embedded program.

`infolist = IDE_Obj.list('function',functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = IDE_Obj.list('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
`infolist.functionname(1).name`	Function name
`infolist.functionname(1).filename`	Name of file where function is defined
`infolist.functionname(1).address`	Relevant address information such as start address and end address
`infolist.functionname(1).funcvar`	Variables local to the function
`infolist.functionname(1).uclass`	`ticcs` object class that matches the type of this symbol — **`function`**
`infolist.functionname(1).funcdecl`	Function declaration — where information such as the function return type is contained
`infolist.functionname(1).islibfunc`	Determine if the library is a function
`infolist.functionname(1).linepos`	Start and end line positions of function
`infolist.functionname(1).funcinfo`	Miscellaneous information about the function
`infolist.functionname(2)...`	...
`infolist.functionname(n)...`	...

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 demonstrate using `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
IDE_Obj.list('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=IDE_Obj.list('project')
```

```
projectinfo =

          name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'
          type: 'project'
    processortype: 'TMS320C67XX'
      srcfiles: [69x1 struct]
      buildcfg: [3x1 struct]
```

See Also

`info`

listsessions

Purpose

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 all 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`

Purpose

Load program file onto processor

Syntax

`IDE_Obj.load(`*`filename`*`,`*`timeout`*`)`

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

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

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

Using load with Eclipse IDE

With Eclipse IDE:

- Before using `load`, use `activate` to make the project associated with the executable file active.
- For the *`filename`* argument, use a relative or absolute path to specify the executable file.

A relative path consists of:

`project/configuration/executablefile`

An absolute path consists of:

`workspace/project/configuration/executablefile`

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

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

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

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

Examples

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

See Also

`cd` | `dir` | `open`

Purpose

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

Syntax

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

Argument

firstTime

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

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

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

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

Description

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

The generated code for a Simulink model calls *model*_initialize once, at the beginning of model execution. If your selected target supports *firstTime* argument control and IncludeERTFirstTime is set to on, the generated code passes in *firstTime* as 1 (TRUE).

See Also

model_SetEventsForThisBaseStep | model_step | model_terminate

How To

- "Entry Point Functions and Scheduling"
- Command Line Information

Purpose

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

Syntax

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

Arguments

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

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

Description

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

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

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

See Also

`model_initialize` | `model_step` | `model_terminate`

How To

- "Entry Point Functions and Scheduling"

Purpose

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

Syntax

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

Arguments

tid

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

Calling Interfaces

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

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

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

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

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

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

Description

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

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

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

- `model_output`: Contains the output code for all blocks in the model
- `model_update`: Contain the update code for all blocks in the model

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

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

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

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

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

See Also

model_initialize | model_SetEventsForThisBaseStep | model_terminate

How To

- "Entry Point Functions and Scheduling"

model_terminate

Purpose

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

Syntax

`void model_terminate(void)`

Description

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

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

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

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

See Also

`model_initialize` | `model_SetEventsForThisBaseStep` | `model_step`

How To

- "Entry Point Functions and Scheduling"

rtw.codegenObjectives.Objective.modifyInheritedParam

Purpose

Modify inherited parameter values

Syntax

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

Description

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

Input Arguments

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

Examples

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

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

See Also

`get_param`

How To

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

msgcount

Purpose

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(IDE_Obj.rtdx,'ichannel', int32(indata));
   ```

3. Use `msgcount` to determine the number of messages available in the queue.

   ```
   num_of_msgs = msgcount(IDE_Obj.rtdx,'ichannel')
   ```

See Also

`read` | `readmat` | `readmsg`

Purpose

Create project, library, or build configuration in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

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

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

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

Examples

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

See Also

`activate` | `close`

Purpose

Open project in IDE

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

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

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

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

Examples

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

See Also

`cd` | `dir` | `load` | `new`

Purpose

Create plot for signal or multiple signals

Syntax

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

Description

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

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

Input Arguments

`dataset`

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

`'Signals', signal_list`

Parameter/value argument pair specifying the signal or signals to plot. The value for this parameter can be an individual signal name, or a cell array of strings, where each string is a signal name in the *`dataset`*. Use `cgv.CGV.getSavedSignals` to view the list of available signal names in the *`dataset`*. The syntax for an individual signal name is:

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

The syntax for a list of signal names is:

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

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

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

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

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

Output Arguments

Depending on the data, any of the following parameters might be empty:

`signal_names`

Cell array of signal names

`signal_figures`

Array of figure handles for signals

How To

- "Numerical Equivalence Checking"

Purpose

Generate real-time execution or stack profiling report

Syntax

IDE_Obj.profile(*type*,*action*,*timeout*)

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

Use *IDE_Obj*.profile(*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	Eclipse IDE	MULTI IDE	VisualDSP++ IDE
'execution' — Execution profiling	Yes	Yes, with limitations.	Yes	Yes
'stack'— Stack profiling	Yes			Yes

Currently, with the Eclipse IDE, you can only perform execution profiling for ARM processors running Linux.

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 "Execution Profiling for Embedded 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 "Stack Profiling for Embedded 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. In a model that has a Target Preferences block, open the model configuration parameters (**Ctrl+ E**).

2. Select the IDE Link pane.

3. Enable **Profile real-time execution**.

4. Build your model.

   ```
   IDE_Obj.build
   ```

5. Load your program to the processor.

   ```
   IDE_Obj.load('c:\work\sumdiff.out')
   ```

6 For stack profiling, initialize the stack to a known state. (For execution profiling, skip this step.)

```
IDE_Obj.profile('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 correctly.

7 Run the program on the processor.

```
IDE_Obj.run
```

8 Stop the running program.

```
IDE_Obj.halt
```

9 To get the profiling reports enter one of the following commands:

```
IDE_Obj.profile('stack','report') #Get stack profiling report
IDE_Obj.profile('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 *nnn* seconds.	Profiling data was recorded over *nnn* 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`

Purpose

Working folder used by Eclipse

Syntax

`wd= IDE_Obj.pwd`

IDEs

This function supports the following IDEs:

- Eclipse IDE

Description

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

Examples

To get the Eclipse IDE working folder:

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

wd =

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

See Also

`dir`

Purpose

Read data from processor memory

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

- The start address
- The memory type

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using read with MULTI

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

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

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

Examples

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

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

`131072` is the decimal address of the data to read.

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

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

See Also

`write`

Purpose

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. If necessary, use the RTDX tools provided in the IDE to determine which channels exist for the loaded program.

`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 properly, 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 all 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 = IDE_Obj.rtdx;
open(rx,'ichannel','w');
enable(rx,'ichannel');
```

```
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
IDE_Obj.write(0,indata,30);
outdata=IDE_Obj.read(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

Purpose

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 all 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 all 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. `readmsgs` 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`. All of the optional input arguments—`nummsgs`, `siz`, and `timeout`—use their default values.

In all 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 all the messages read from the channel to that point.

Examples

```
IDE_Obj = ticcs;
rx = IDE_Obj.rtdx;
open(rx,'ichannel','w');
enable(rx,'ichannel');
open(rx,'ochannel','r');
enable(rx,'ochannel');
indata = 1:25; % Set up some data.
IDE_Obj.write(0,indata,30);
outdata=IDE_Obj.read(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 = IDE_Obj.rtdx.readmsg('ochannel','double',[4 5])
```

See Also

`read` | `readmat` | `writemsg`

rtw.codegenObjectives.Objective.register

Purpose

Register objective

Syntax

`register(obj)`

Description

`register(obj)` registers *`obj`* Register and add *`obj`* to the end of the list of available objectives that you can use with the Code Generation Advisor.

Input Arguments

`obj`	Handle to a code generation objective object previously created.

Examples

Register the objective:

```
register(obj);
```

See Also

`DAStudio.CustomizationManager.ObjectiveCustomizer`

How To

- “Creating Custom Objectives”
- “Registering Customizations”

Purpose

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

Syntax

```
entry = registerCFunctionEntry(hTable, priority,
                               numInputs, functionName,
                               inputType, implementationName,
                               outputType, headerFile,
                               genCallback, genFileName)
```

Input Arguments

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

priority
: Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

numInputs
: Positive integer specifying the number of input arguments.

functionName
: String specifying the name of the function to be replaced. The name must match one of the functions supported for replacement:

Math Functions

Note For detailed support information, see "Example: Mapping Math Functions to Target-Specific Implementations".

abs	acos	acosh	asin
asinh	atan	atan2	atanh

ceil	cos	cosh	exactrSqrt
exp	fix	floor	frexp
hypot	ldexp	ln	log
log10	max	min	mod/fmod
pow	rem	round	rSqrt
saturate	sign	sin	sincos
sinh	sqrt	round	tanh
Memory Utility Functions			
memcmp	memcpy	memset	memset2zero[1]
Nonfinite Support Utility Functions[2]			
getInf	getMinusInf	getNaN	isInf[3]
isNaN[3]			

Notes:
[1] Some target processors provide optimized `memset` functions for use when performing a memory set to zero. The TFL API supports replacing `memset` to zero functions with more efficient target-specific functions.
[2] Replacement of nonfinite functions is supported for Simulink code generation (not for Stateflow® or MATLAB® Coder™ code generation).
[3] Replacement of `isInf` and `isNaN` is supported only for complex floating-point inputs.

inputType
String specifying the data type of the input arguments, for example, `'double'`. (This function requires that all input arguments are of the same type.)

implementationName
String specifying the name of your implementation. For example, if *functionName* is `'sqrt'`, *implementationName* can be `'sqrt'` or a different name of your choosing.

outputType
String specifying the data type of the return argument, for example, `'double'`.

headerFile
String specifying the header file in which the implementation function is declared, for example, `'<math.h>'`.

genCallback
String specifying `''` or `'RTW.copyFileToBuildDir'`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Code Replacements" in the Embedded Coder documentation.

genFileName
String specifying `''`. (This argument is for use only by MathWorks developers.)

Output Arguments

Handle to the created TFL function entry. Specifying the return argument in the `registerCFunctionEntry` function call is optional.

Description

The `registerCFunctionEntry` function provides a quick way to create and register a TFL function entry. This function can be used only if your TFL function entry meets the following conditions:

- All input arguments are of the same type.
- All input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*`n`*
  - For return argument, `y1`

Examples

In the following example, the `registerCFunctionEntry` function is used to create a function entry for `sqrt` in a TFL table.

```
hLib = RTW.TflTable;

hLib.registerCFunctionEntry(100, 1, 'sqrt', 'double', 'sqrt', ...
                              'double', '<math.h>', '', '');
```

See Also

`registerCPromotableMacroEntry`

How To

- “Alternative Method for Creating Function Entries”
- “Creating Code Replacement Tables”
- “Code Replacement”

Purpose

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

Syntax

```
entry = registerCPPFunctionEntry(hTable, priority,
                                 numInputs, functionName,
                                 inputType, implementationName,
                                 outputType, headerFile,
                                 genCallback, genFileName,
                                 nameSpace)
```

Input Arguments

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

priority
: Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

numInputs
: Positive integer specifying the number of input arguments.

functionName
: String specifying the name of the function to be replaced. The name must match one of the functions supported for replacement:

Math Functions

Note For detailed support information, see "Example: Mapping Math Functions to Target-Specific Implementations".

abs	acos	acosh	asin
asinh	atan	atan2	atanh

ceil	cos	cosh	exactrSqrt
exp	fix	floor	frexp
hypot	ldexp	ln	log
log10	max	min	mod/fmod
pow	rem	round	rSqrt
saturate	sign	sin	sincos
sinh	sqrt	round	tanh
Memory Utility Functions			
memcmp	memcpy	memset	memset2zero[1]
Nonfinite Support Utility Functions[2]			
getInf	getMinusInf	getNaN	isInf[3]
isNaN[3]			

Notes:
[1] Some target processors provide optimized `memset` functions for use when performing a memory set to zero. The TFL API supports replacing `memset` to zero functions with more efficient target-specific functions.
[2] Replacement of nonfinite functions is supported for Simulink code generation (not for Stateflow or MATLAB Coder code generation).
[3] Replacement of `isInf` and `isNaN` is supported only for complex floating-point inputs.

inputType
String specifying the data type of the input arguments, for example, `'double'`. (This function requires that all input arguments are of the same type.)

implementationName
String specifying the name of your implementation. For example, if *functionName* is `'sqrt'`, *implementationName* can be `'sqrt'` or a different name of your choosing.

outputType
: String specifying the data type of the return argument, for example, `'double'`.

headerFile
: String specifying the header file in which the implementation function is declared, for example, `'<math.h>'`.

genCallback
: String specifying `''` or `'RTW.copyFileToBuildDir'`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Code Replacements" in the Embedded Coder documentation.

genFileName
: String specifying `''`. (This argument is for use only by MathWorks developers.)

nameSpace
: String specifying the C++ name space in which the implementation function is defined. If this function entry is matched, the software emits the name space in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`). If you specify `''`, the software does not emit a name space designation in the generated code.

Output Arguments

Handle to the created TFL C++ function entry. Specifying the return argument in the `registerCPPFunctionEntry` function call is optional.

Description

The `registerCPPFunctionEntry` function provides a quick way to create and register a TFL C++ function entry. This function can be used only if your TFL C++ function entry meets the following conditions:

- All input arguments are of the same type.

- All input argument names and the return argument name follow the default Simulink naming convention:
  - For input argument names, `u1`, `u2`, ..., `u`*n*
  - For return argument, `y1`

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

Examples

In the following example, the `registerCPPFunctionEntry` function is used to create a C++ function entry for `sin` in a TFL table.

```
hLib = RTW.TflTable;

hLib.registerCPPFunctionEntry(100, 1, 'sin', 'single', 'sin', ...
                              'single', 'cmath', '', '', 'std');
```

See Also

`enableCPP` | `setNameSpace`

How To

- “Alternative Method for Creating Function Entries”
- “Creating Code Replacement Tables”
- “Code Replacement”

Purpose

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

Syntax

```
entry = registerCPromotableMacroEntry(hTable, priority,
                                      numInputs, functionName,
                                      inputType, implementationName,
                                      outputType, headerFile,
                                      genCallback, genFileName)
```

Input Arguments

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

priority
: Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

numInputs
: Positive integer specifying the number of input arguments.

functionName
: String specifying the name of the function to be replaced. Specify `'abs'`. (This function should be used only for `abs` function replacement.)

inputType
: String specifying the data type of the input arguments, for example, `'double'`. (This function requires that all input arguments are of the same type.)

implementationName
: String specifying the name of your implementation. For example, assuming *functionName* is `'abs'`, *implementationName* can be `'abs'` or a different name of your choosing.

outputType
String specifying the data type of the return argument, for example, `'double'`.

headerFile
String specifying the header file in which the implementation function is declared, for example, `'<math.h>'`.

genCallback
String specifying `''` or `'RTW.copyFileToBuildDir'`. If you specify `'RTW.copyFileToBuildDir'`, and if this function entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Code Replacements" in the Embedded Coder documentation.

genFileName
String specifying `''`. (This argument is for use only by MathWorks developers.)

Output Arguments

Handle to the created TFL promotable macro entry. Specifying the return argument in the `registerCPromotableMacroEntry` function call is optional.

Description

The `registerCPromotableMacroEntry` function creates a TFL promotable macro entry based on specified parameters and registers the entry in the TFL table. A promotable macro entry will promote the output data type based on the target word size.

This function provides a quick way to create and register a TFL promotable macro entry. This function can be used only if your TFL function entry meets the following conditions:

- All input arguments are of the same type.
- All input argument names and the return argument name follow the default Simulink naming convention:

- For input argument names, `u1`, `u2`, ..., `un`
- For return argument, `y1`

Note This function should be used only for `abs` function replacement. Other functions supported for replacement should use `registerCFunctionEntry`.

Examples

In the following example, the `registerCPromotableMacroEntry` function is used to create a function entry for `abs` in a TFL table.

```
hLib = RTW.TflTable;

hLib.registerCPromotableMacroEntry(100, 1, 'abs', 'double', 'abs_prime', ...
                                   'double', '<math_prime.h>', '', '');
```

See Also

`registerCFunctionEntry`

How To

- "Alternative Method for Creating Function Entries"
- "Creating Code Replacement Tables"
- "Code Replacement"

regread

Purpose

Values from processor registers

Syntax

```
reg=IDE_Obj.regread('regname','represent',timeout)
reg = IDE_Obj.regread('regname','represent')
reg = IDE_Obj.regread('regname')
```

IDEs

This function supports the following IDEs:

- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

`reg=IDE_Obj.regread('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, ISTP, 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 `represent` 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 = IDE_Obj.regread('regname','represent')` does not set the global time-out value. The time-out value in `IDE_Obj` applies.

`reg = IDE_Obj.regread('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 at any time 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 consisten, 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 MULTI IDE

For the MPC5554 processor, most registers are memory-mapped and consequently are available using `read` and `write`. However, use `regread` to read the PC register. The following command demonstrates how to read the PC register. To identify the target, `IDE_Obj` is the IDE handle.

```
IDE_Obj.regread('PC','binary')
```

To tell MATLAB what data type you are reading, the string `binary` indicates that the `PC` register contains a value stored as an unsigned binary integer.

In response, MATLAB displays

```
ans =

        33824
```

For processors in the Blackfin family, `regread` lets you access processor registers directly. To read the value in general purpose register cycles, type the following function.

```
treg = IDE_Obj.regread('cycles','2scomp');
```

`treg` now contains the two's complement representation of the value in `A0`.

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 demonstrates how to read the PC register. To identify the processor, `IDE_Obj` is a link for CCS IDE.

```
IDE_Obj.regread('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 = IDE_Obj.regread('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

```
IDE_Obj.regread('B2','binary');
```

See Also

`read` | `regwrite` | `write`

Purpose

Write data values to registers on processor

Syntax

```
IDE_Obj.regwrite('regname',value,'represent',timeout)
IDE_Obj.regwrite('regname',value,'represent')
IDE_Obj.regwrite('regname',value,)
```

IDEs

This function supports the following IDEs:

- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

IDE_Obj.regwrite('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 value to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the represent argument.
'binary'	Write value to the destination register as an unsigned binary integer.
'ieee'	Write value 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`, `ISTP`, `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.

`IDE_Obj``.regwrite('regname',value,'represent')` omits the `timeout` input argument and does not change the time-out value specified in `IDE_Obj`.

`IDE_Obj``.regwrite('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 at any time 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

```
IDE_Obj.regwrite('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.

```
IDE_Obj.regwrite('b1:b0',hex2dec('1010'),'ieee')
```

Registers B1:B0 now contain the value 4112 in double-precision format.

See Also

read | regread | write

Purpose

Reload most recent program file to processor signal processor

Syntax

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

IDEs

This function supports the following IDEs:

- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

Using reload with Multiprocessor Boards

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

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

Examples

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

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

s =

     ''

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

IDE_Obj.build

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

s =

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

See Also

`cd` | `load` | `open`

Purpose

Build Simulink-generated code on remote target running Linux

Syntax

```
remoteBuild(buildinfo,targetrtwstartdir,targetipaddress,
    username,passwd,putilsfolder)
```

Description

`remoteBuild(buildinfo,targetrtwstartdir,targetipaddress, username,passwd,putilsfolder)` builds your generated code on a remote Linux target. This approach enables a developer using a Windows host computer to build an application from generated code in Linux on the target environment.

Tips

- The host must be running Windows. Install ssh and scp utilities, such as plink.exe and pscp.exe, on this Windows host. These utilities are available from the PuTTY download page at http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
- The remote target must be running Linux, with ssh and scp protocols enabled and GCC-based compiler, linker, and archiver tools installed.
- The remote build is a two-stage process. In the first stage, you generate source files and makefile for a Simulink model without compiling and linking. In the second stage, you run `remoteBuild` to transfer the generated files to the remote target, where the compiler and linker complete the build process. Also see "Example: Build Generated Code on a BeagleBoard Running Linux".

Input Arguments

`buildinfo`

Specify the object that contains the build information structure of the model. For example, `bd.buildInfo`.

First, use the Simulink `load` command to create this object from the buildInfo.mat file, located among the files you generated from your model. For example,

```
bd = load('C:\Documents\MATLAB\foo_eclipseide\buildInfo.mat')
```

`targetrtwstartdir`

The path of the destination folder on the remote Linux target to which `remoteBuild` copies the generated source and header files. For example: `'/home/root/devel'`

If the destination folder does not exist, `remoteBuild` creates it.

`targetipaddress`

The IP address or the host name of the remote Linux target. For example, `'10.10.10.1'`

`username`

The name of the user that runs ssh commands on the remote Linux target. For example, `'root'`

`passwd`

Enter the password for `username`. If the username does not have a password, provide empty quotes. For example `''`

`putilsfolder`

The path of the folder on the Windows host that contains plink.exe and pscp.exe. For example, `'C:\putils'`

Examples

Using the examples in the preceding input arguments, the resulting command would be:

```
bd = load('C:\Documents\MATLAB\foo_eclipseide\buildInfo.mat')
remoteBuild(bd.buildInfo,'/home/root/devel','10.10.10.1','root','','C:\putils')
```

See Also

`xmakefilesetup | load`

Purpose

Remove file, project, or breakpoint

Syntax

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

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

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

Input Arguments

`IDE_Obj`

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

`filename`

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

`filetype`

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

Default: `'text'`

`addr`

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

`debugtype`

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

Default: `'break'` (breakpoint)

`line`

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

`timeout`

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

Examples

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

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

Now remove one file from your project

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

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

See Also

add | cd | open

RTW.AutosarInterface.removeEventConf

Purpose

Remove AUTOSAR event from model

Syntax

`autosarInterfaceObj.removeEventConf(EventName)`

Description

`autosarInterfaceObj``.removeEventConf(`*`EventName`*`)` removes *`EventName`* from *`autosarInterfaceObj`*, a model-specific RTW.AutosarInterface object.

Input Arguments

`EventName`

Name of AUTOSAR RTEEvent

See Also

`RTW.AutosarInterface.addEventConf`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

rtw.codegenObjectives.Objective.removeInheritedCheck

Purpose

Remove inherited checks

Syntax

`removeInheritedCheck(obj, checkID)`

Description

`removeInheritedCheck(obj, checkID)` removes an inherited check from the objective definition. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

Input Arguments

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

Examples

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

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

See Also

`Simulink.ModelAdvisor`

How To

- "Creating Custom Objectives"
- "About IDs"

rtw.codegenObjectives.Objective.removeInheritedParam

Purpose

Remove inherited parameters

Syntax

`removeInheritedParam(obj, paramName)`

Description

`removeInheritedParam(obj, paramName)` removes an inherited parameter from this objective. Use this method when you create a new objective from an existing objective.

When the user selects multiple objectives, if another objective includes the parameter, the Code Generation Advisor reviews the parameter value using **Check model configuration settings against code generation objectives**.

Input Arguments

obj	Handle to a code generation objective object previously created.
paramName	Parameter that you want to remove from the objective.

Examples

Remove `Inlineparameters` from the objective.

```
removeInheritedParam(obj, 'InlineParams');
```

See Also

`get_param`

How To

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

Purpose Open code execution profiling report

Syntax *myExecutionProfile*.report

Description *myExecutionProfile*.report opens the code execution profiling report.

myExecutionProfile is a workspace variable generated by a SIL or PIL simulation.

See Also display

How To

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"

Purpose

Stop program execution and reset processor

Syntax

IDE_Obj.reset(*timeout*)

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

IDE_Obj.reset(*timeout*) stops the program executing on the processor and asynchronously performs a processor reset, returning all 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`

Purpose

Reload most recent program file to processor signal processor

Syntax

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

IDEs

This function supports the following IDEs:

- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

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

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

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

See Also

halt | isrunning | run

rtIOStreamClose

Purpose

Shut down communications channel with remote processor

Syntax

```
int rtIOStreamClose(
    int streamID
)
```

Arguments

streamID
A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

Description

```
int rtIOStreamClose(
    int streamID
)
```

Call this function to shut down the communications channel and clean up any associated resources.

A return value of zero indicates success. `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtiostream_wrapper`

How To

- "Creating a Connectivity Configuration for a Target"
- rtwdemo_rtiostream
- rtwdemo_custom_pil

rtIOStreamOpen

Purpose

Initialize communications channel with remote processor

Syntax

```
int rtIOStreamOpen(
     int    argc,
     void * argv[ ]
)
```

Arguments

argc

Integer argument count, i.e., the number of parameters in `argv[]`

argv[]

An array of pointers to parameters; typically these are null-terminated string parameters, however, this is allowed to be implementation dependent.

Description

```
int rtIOStreamOpen(
     int    argc,
     void * argv[ ]
)
```

This function initializes a communication stream to allow exchange of data between host and target.

The input parameters allows driver-specific parameters to be passed to the communications driver.

If successful, the function returns a nonnegative integer greater than zero, representing a stream handle. A return value of `RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See Also

`rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose` | `rtiostream_wrapper`

How To

- “Creating a Connectivity Configuration for a Target”
- rtwdemo_rtiostream
- rtwdemo_custom_pil

Purpose

Receive data from remote processor

Syntax

```
int rtIOStreamRecv(
    int      streamID,
    void   * dst,
   size_t   size,
    size_t * sizeRecvd
)
```

Arguments

streamID
: A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

size
: Size of data to copy into the buffer. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

dst
: A pointer to the start of the buffer where received data must be copied.

sizeRecvd
: The number of units of data received and copied into the buffer `dst` (zero if no data was copied).

Description

```
int rtIOStreamRecv(
    int      streamID,
    void   * dst,
    size_t   size,
    size_t * sizeRecvd
)
```

This function receives data over a communication channel with a remote processor.

A return value of zero indicates success.`RTIOSTREAM_ERROR` indicates an error.

`RTIOSTREAM_ERROR` is defined in `rtiostream.h` as:

```
#define RTIOSTREAM_ERROR (-1)
```

See also `rtiostreamSend` for implementation and performance considerations.

See Also

`rtIOStreamSend` | `rtIOStreamOpen` | `rtIOStreamClose` | `rtIOStream_wrapper`

How To

- "Creating a Connectivity Configuration for a Target"
- rtwdemo_rtiostream
- rtwdemo_custom_pil

Purpose Send data to remote processor

Syntax

```
int rtIOStreamSend(
    int          streamID,
   const void * src,
    size_t       size,
   size_t     * sizeSent
)
```

Arguments

streamID
: A handle to the stream that was returned by a previous call to `rtIOStreamOpen`.

src
: A pointer to the start of the buffer containing an array of data to transmit

size
: Size of data to transmit. For byte-addressable architectures, size is measured in bytes. Some DSP architectures are not byte-addressable. In these cases, size is measured in number of WORDs, where `sizeof(WORD) == 1`.

sizeSent
: Size of data actually transmitted (always less than or equal to `size`), or zero if no data was transmitted

Description

```
int rtIOStreamSend(
    int          streamID,
    const void * src,
    size_t       size,
    size_t     * sizeSent
)
```

This function sends data over a communication stream with a remote processor.

A return value of zero indicates success.RTIOSTREAM_ERROR indicates an error.

RTIOSTREAM_ERROR is defined in rtiostream.h as:

```
#define RTIOSTREAM_ERROR (-1)
```

Implementation and Performance Considerations

The API for rtIOStream functions is designed to be independent of the physical layer across which the data is sent. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN). The choice of physical layer affects the achievable data rates for the host-target communication.

For a processor-in-the-loop (PIL) application there is no minimum data rate requirement. However, the higher the data rate, the faster the simulation will run.

In general, a communications device driver will require additional hardware-specific or channel-specific configuration parameters. For example:

- A CAN channel may require specification of which available CAN Node should be used.
- A TCP/IP channel may require a port or static IP address to be configured.
- A CAN channel may require the CAN message ID and priority to be specified.

It is the responsibility of the user who implements the rtIOStream driver functions to provide this configuration data, for example by hard-coding it, or by supplying arguments to rtIOStreamOpen.

See Also

rtIOStreamOpen | rtIOStreamClose | rtIOStreamRecv | rtiostream_wrapper

How To

- “Creating a Connectivity Configuration for a Target”
- rtwdemo_rtiostream
- rtwdemo_custom_pil

rtiostream_wrapper

Purpose

Test rtiostream shared library methods

Syntax

```
STATION_ID = rtiostream_wrapper(SHARED_LIB,'open')
STATION_ID =
rtiostream_wrapper(SHARED_LIB,'open',p1, v1, p2,
   v2, ...)
[RES,SIZE_SENT] = rtiostream_wrapper(SHARED_LIB,'send',ID,
   DATA, SIZE)
[RES, 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 any 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.

Shared library can be:

- *`libTCPIP`* — For TCP/IP communication. Value depends on your operating system. See `rtwdemo_rtiostream`.
- `'rtiostreamserial.dll'` — For serial communication, Windows only.

`open`

Opens communication channel

`send`

Sends data into communication channel with handle *`ID`*

`ID`

Communication channel handle

`DATA`

Data to be sent

`SIZE`

Size of requested data in bytes

`recv`

Receives data from communication channel with handle *`ID`*

`close`

Closes communication channel with handle *`ID`* `unloadlibrary`

Unloads *`SHARED_LIB`*

Name-Value Pair Arguments

`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`

`-client`

- 0 — Opens as TCP/IP server
- 1 — Opens as TCP/IP client

Shared library must be *`libTcpip`*.

`-port`

Port number for TCP/IP or COM port string for serial communication. If port is for serial communication, you must also specify bit rate using `-baud`.

Shared library must be either *`libTcpip`* or `'rtiostreamserial.dll'`.

`-hostname`

Identifier for host computer, for example, `'localhost'`.

Shared library must be *`libTcpip`*.

`-baud`

Bit rate for serial communication port.

Shared library must be `'rtiostreamserial.dll'`.

Output Arguments

STATION_ID

Handle to communication channel if attempt to open channel is successful. 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. May be less than *SIZE*, that is, the requested number of bytes to send.

DATA_RECVD

Data received

SIZE_RECVD

Number of bytes actually received from channel. May be less than *SIZE*, that is, the requested number of bytes to send.

Examples

The following examples open communication channels using supplied TCP/IP and serial communication drivers.

The following command opens `rtiostream` channel `stationA` as a TCP/IP server:

```
stationA = rtiostream_wrapper('rtiostreamtcpip.dll','open',...
                              '-client', '0',...
                              '-port', port_number);
```

The following command opens the rtiostream channel `StationB` as a TCP/IP client:

```
stationB = rtiostream_wrapper('rtiostreamtcpip.dll','open',...
                              '-client','1',...
                              '-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), you must specify the bit rate when you open a channel with a specific port. Specify the option `'-baud'` with a value for the bit rate. For example, the following command opens `COM1` with a bit rate of 9600:

```
stationA = rtiostream_wrapper('rtiostreamserial.dll','open',...
                              '-port','COM1',...
                              '-baud','9600');
```

See Also

`rtIOStreamOpen` | `rtIOStreamSend` | `rtIOStreamRecv` | `rtIOStreamClose`

How To

- "Creating a Connectivity Configuration for a Target"
- rtwdemo_rtiostream
- rtwdemo_custom_pil

Purpose

Control and validate AUTOSAR configuration

Description

You can use methods of the `RTW.AutosarInterface` class to configure AUTOSAR code generation and XML import and export options.

Construction

RTW.AutosarInterface	Construct `RTW.AutosarInterface` object

Methods

addEventConf	Add configured AUTOSAR event to model
addIOConf	Add AUTOSAR I/O configuration to model
attachToModel	Attach `RTW.AutosarInterface` object to model
getArxmlFilePackaging	Get AUTOSAR XML packaging format
getComponentName	Get XML component name
getComponentType	Get type of software component
getDataTypePackageName	Get XML data type package name
getDefaultConf	Get default configuration
getEventType	Get event type
getExecutionPeriod	Get runnable execution period
getImplementationName	Get name of XML implementation
getInitEventName	Get initial event name
getInitRunnableName	Get initial runnable name
getInterfacePackageName	Get XML interface package name

getInternalBehaviorName	Get name of XML file that specifies software component internal behavior
getIOAutosarPortName	Get I/O AUTOSAR port name
getIODataAccessMode	Get I/O data access mode
getIODataElement	Get I/O data element name
getIOErrorStatusReceiver	Get name of error status receiver port
getIOInterfaceName	Get I/O interface name
getIOPortNumber	Get I/O AUTOSAR port number
getIOServiceInterface	Get port I/O service interface
getIOServiceName	Get port I/O service name
getIOServiceOperation	Get port I/O service operation
getIsServerOperation	Determine whether server is specified
getPeriodicEventName	Get periodic event name
getPeriodicRunnableName	Get periodic runnable name
getServerInterfaceName	Get name of server interface
getServerOperationPrototype	Get server operation prototype
getServerPortName	Get server port name
getServerType	Determine server type
getTriggerPortName	Get name of Simulink inport that provides trigger data for DataReceivedEvent
removeEventConf	Remove AUTOSAR event from model
runValidation	Validate `RTW.AutosarInterface` object against model

setArxmlFilePackaging	Set AUTOSAR XML packaging format
setComponentName	Set XML component name
setComponentType	Set type of software component
setDataTypePackageName	Specify XML package name for data type
setEventType	Set type for event
setExecutionPeriod	Specify execution period for TimingEvent
setImplementationName	Set name of XML implementation
setInitEventName	Set initial event name
setInitRunnableName	Set initial runnable name
setInterfacePackageName	Set name of XML interface package
setInternalBehaviorName	Set name of XML file for software component internal behavior
setIOAutosarPortName	Set AUTOSAR port name
setIODataAccessMode	Set I/O data access mode
setIODataElement	Set I/O data element
setIOErrorStatusReceiver	Set name of error status receiver port
setIOInterfaceName	Set I/O interface name
setIOServiceInterface	Set port I/O service interface
setIOServiceName	Set port I/O service name
setIOServiceOperation	Set port I/O service operation
setIsServerOperation	Indicate that server is specified
setPeriodicEventName	Set periodic event name

setPeriodicRunnableName	Set periodic runnable name
setServerInterfaceName	Set name of server interface
setServerOperationPrototype	Specify operation prototype
setServerPortName	Set server port name
setServerType	Specify server type
setTriggerPortName	Specify Simulink inport that provides trigger data for DataReceivedEvent
syncWithModel	Synchronize configuration with model

Copy Semantics

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

How To

- “Using the Configure AUTOSAR Interface Dialog Box”
- “Configuring Ports for Basic Software and Error Status Receivers”
- “Modifying and Validating an Existing AUTOSAR Interface”

RTW.AutosarInterface

Purpose

Construct `RTW.AutosarInterface` object

Syntax

```
autosarInterfaceObject = RTW.AutosarInterface()
autosarInterfaceObject = RTW.AutosarInterface(model_handle)
autosarInterfaceObject = RTW.AutosarInterface(object_name,
   model_handle)
```

Description

`autosarInterfaceObject` `= RTW.AutosarInterface()` creates an `RTW.AutosarInterface` object without specifying a model, and returns a handle to this object.

`autosarInterfaceObject` `= RTW.AutosarInterface(`*`model_handle`*`)` creates an `RTW.AutosarInterface` object with a model specified, and returns a handle to this object. The software sets the name of the `RTW.AutosarInterface` object to `'AutosarInterface'`.

`autosarInterfaceObject` `= RTW.AutosarInterface(`*`object_name`*`,` *`model_handle`*`)` creates an `RTW.AutosarInterface` object with a model specified, and returns a handle to this object. The software sets the name of the `RTW.AutosarInterface` object to *`object_name`*.

Input Arguments

`model_handle`	Handle to Simulink model
`object_name`	Name of `RTW.AutosarInterface` object

Output Arguments

`autosarInterfaceObject`	Handle to newly created `RTW.AutosarInterface` object.

How To

- "Generating Code for AUTOSAR Software Components"
- `RTW.AutosarInterface.attachToModel`

rtw.codegenObjectives.Objective

Purpose

Customize code generation objectives

Description

An `rtw.codegenObjectives.Objective` object creates a code generation objective.

Construction

rtw.codegenObjectives.Objective	Create custom code generation objectives

Methods

addCheck	Add checks
addParam	Add parameters
excludeCheck	Exclude checks
modifyInheritedParam	Modify inherited parameter values
register	Register objective
removeInheritedCheck	Remove inherited checks
removeInheritedParam	Remove inherited parameters
setObjectiveName	Specify objective name

Copy Semantics

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

Examples

Create a custom objective named `Reduce RAM Example`. The following code is the contents of the `sl_customization.m` file that you create.

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();
```

```
end

function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');

%Register the objective
register(obj);

end
```

See Also

DAStudio.CustomizationManager.ObjectiveCustomizer

How To

- “Creating Custom Objectives”

rtw.codegenObjectives.Objective

Purpose

Create custom code generation objectives

Syntax

```
obj = rtw.codegenObjectives.Objective('objID')
obj = rtw.codegenObjectives.Objective('objID',
'base_objID')
```

Description

`obj = rtw.codegenObjectives.Objective('objID')` creates an objective object, `obj`.

`obj = rtw.codegenObjectives.Objective('objID', 'base_objID')` creates an object, `obj`, for a new objective that is identical to an existing objective. You can then modify the new objective to meet your requirements.

Input Arguments

objID	A permanent, unique identifier for the objective. • You must have *objID*. • The value of *objID* must remain constant. • When you refresh your customizations, if *objID* is not unique, Simulink generates an error.
base_objID	The identifier of the objective that you want to base the new objective on.

Examples

Create a new objective:

```
obj = rtw.codegenObjectives.Objective('ex_ram_1');
```

Create a new objective based on the existing `Execution efficiency` objective:

```
obj = rtw.codegenObjectives.Objective('ex_my_efficiency_1', 'Execution efficiency');
```

How To

- "Creating Custom Objectives"

RTW.configSubsystemBuild

Purpose

Configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem

Syntax

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

Description

`RTW.configSubsystemBuild(`*`block`*`)` opens a graphical user interface where you can configure either C function prototype information or C++ encapsulation interface information for right-click builds of a specified nonvirtual subsystem. The appropriate dialog box opens based on the **Language** value selected for your model on the **Code Generation** pane of the Configuration Parameters dialog box.

To configure and generate C++ encapsulation interfaces for a nonvirtual subsystem, you must

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

Input Arguments

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

How To

- "Configuring Function Prototypes for Nonvirtual Subsystems"
- "Function Prototype Control"
- "Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems"
- "C++ Encapsulation Interface Control"

Purpose

Provide parameters to each target connectivity component

Syntax

```
componentArgs = rtw.connectivity.ComponentArgs (componentPath,
   componentCodePath, componentCodeName, applicationCodePath)
```

Description

Syntax of constructor `ComponentArgs`:

```
componentArgs = rtw.connectivity.ComponentArgs
(componentPath, componentCodePath, componentCodeName,
applicationCodePath)
```

You can use the methods of this class to get information about the source component (e.g., the referenced model under test) and the target application (e.g., the PIL application).

For methods, see the following table.

Method	Syntax and Description
getComponentPath	`componentPath = obj.getComponentPath`
	Returns the Simulink system path of the source component (e.g., the path of the referenced model that is under test).
getComponentCodePath	`componentCodePath = obj.getComponentCodePath`
	Returns the Embedded Coder code generation directory path associated with the source component (e.g., the code generation directory of the referenced model that is under test).

Method	Syntax and Description
getComponentCodeName	`componentCodeName = obj.getComponentCodeName`
	Returns the component name used for code generation.
getApplicationCodePath	`applicationCodePath = obj.getApplicationCodePath`
	Returns the directory path associated with the target application (e.g., the path associated with the PIL application).

See `rtw.connectivity.Config` for more information.

See Also

`rtw.connectivity.Config`

How To

- "Verification"
- "Creating a Connectivity Configuration for a Target"

Purpose

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 (e.g. rtw.connectivity.MakefileBuilder) object.
launcher	rtw.connectivity.Launcher object.
communicator	rtw.connectivity.Communicator (e.g. rtw.connectivity.-RtIOStreamHostCommunicator) object.

Constructor syntax:

```
rtw.connectivity.Config(componentArgs, builder, launcher,
communicator)
```

To define a connectivity implementation:

1. You must create a subclass of `rtw.connectivity.Config` that creates instances of your connectivity component classes:
   - rtw.connectivity.MakefileBuilder

- rtw.connectivity.Launcher
- rtw.connectivity.RtIOStreamHostCommunicator

You can see an example `ConnectivityConfig.m`, used in the demo `rtwdemo_custom_pil`.

2 Define the constructor for your subclass as follows:

```
function this = MyConfig(componentArgs)
```

When Simulink creates an instance of your subclass of `rtw.connectivity.Config`, it provides an instance of the `rtw.connectivity.ComponentArgs` class as the only constructor argument. If you want to test your subclass of `rtw.connectivity.Config` manually, you may want to create an `rtw.connectivity.ComponentArgs` object to pass as a constructor argument.

3 After instantiating the builder, launcher and communicator objects in your subclass, call the constructor of the superclass `rtw.connectivity.Config` to define your complete target connectivity configuration, as shown in this example.

```
% call super class constructor to register components
this@rtw.connectivity.Config(componentArgs,...
builder, launcher, communicator);
```

You will register your subclass name (e.g. "`MyPIL.ConnectivityConfig`") to Simulink by using the class `rtw.connectivity.ConfigRegistry`. This uses the `sl_customization.m` mechanism to register your connectivity configuration.

The PIL infrastructure instantiates your subclass as required. The `sl_customization.m` mechanism helps in specifying a suitable connectivity configuration for use with a particular PIL component (and its configuration set). It is also possible for the subclass to do extra validation on construction. For example, you can use the `componentPath` returned by the `getComponentPath` method of the `componentArgs` constructor argument to query and validate parameters associated with the PIL component under test.

For supported hardware implementation settings and other support information, see "SIL and PIL Simulation Support and Limitations" in the Embedded Coder documentation.

See Also

`rtw.connectivity.MakefileBuilder` | `rtw.connectivity.Launcher` | `rtw.connectivity.RtIOStreamHostCommunicator` | `rtw.connectivity.ComponentArgs`

How To

- "Verification"
- "Creating a Connectivity Configuration for a Target"
- rtwdemo_custom_pil

Purpose

Register connectivity configuration

Syntax

```
config = rtw.connectivity.ConfigRegistry
config = rtw.connectivity.ConfigRegistry
```

Description

Use this class to register your connectivity configuration with Simulink by using the `sl_customization.m` mechanism. The connectivity configuration is registered by a call to `registerTargetInfo` inside a `sl_customization.m` file.

Create or add to your `sl_customization.m` file as shown in the "Examples" on page 3-364 section, and place the file on the MATLAB path. Simulink software reads the `sl_customization.m` when it starts, and registers your connectivity configuration. This step also defines the set of Simulink models that the new connectivity configuration is compatible with.

A connectivity configuration must have a unique name and be associated with a connectivity implementation class (a subclass of `rtw.connectivity.Config`). The properties of the configuration (e.g. `SystemTargetFile`) define the set of Simulink models that the connectivity implementation class is compatible with. The properties are shown in the following table.

Properties of rtw.connectivity.ConfigRegistry

Property Name	Description
ConfigName	Unique string name for this configuration
ConfigClass	Full class name of the connectivity implementation (e.g. `rtw.pil.myConnectivityConfig`) to register.

Properties of rtw.connectivity.ConfigRegistry (Continued)

Property Name	Description
SystemTargetFile	Cell array of strings listing System Target Files that support this `ConfigRegistry`. An empty cell array matches any System Target File. The model's `SystemTargetFile`Configuration Parameter is validated against this cell array to determine if this `ConfigRegistry` is valid for use.
TemplateMakefile	Cell array of strings listing Template Makefiles that support this `ConfigRegistry`. An empty cell array matches any Template Makefile and nonmakefile based targets (`GenerateMakefile`: off). The model's `TemplateMakefile` Configuration Parameter is validated against this cell array to determine if this `ConfigRegistry` is valid for use.
TargetHWDeviceType	Cell array of strings listing Hardware Device Types that support this `ConfigRegistry`. An empty cell array matches any Hardware Device Type. The model's `TargetHWDeviceType`Configuration Parameter is validated against this cell array to determine if this `ConfigRegistry` is valid for use.

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.
% $Revision: 1.1.8.6 $

cm.registerTargetInfo(@loc_createConfig);

% local function
function config = loc_createConfig

config = rtw.connectivity.ConfigRegistry;
config.ConfigName = 'My PIL Example';
config.ConfigClass = 'mypil.ConnectivityConfig';

% match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};
% match the standard ert TMF's
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vc.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};
% match regular 32-bit machines and Custom for e.g. ...
% 64-bit Linux
config.TargetHWDeviceType = {'Generic->32-bit x86 ...
                              compatible'
                             'Generic->Custom'};
```

You must configure the file to perform the following steps when Simulink software starts:

1 Create an instance of the `rtw.connectivity.ConfigRegistry` class. For example,

```
config = rtw.connectivity.ConfigRegistry;
```

2 Assign a connectivity configuration name to the `ConfigName` property of the object. For example,

```
config.ConfigName = 'My PIL Example';
```

3 Associate the connectivity configuration with the connectivity API implementation (created in step 1). For example,

```
config.ConfigClass = 'mypil.ConnectivityConfig';
```

4 Define compatible models for this target connectivity configuration, by setting the `SystemTargetFile`, `TemplateMakefile` and `TargetHWDeviceType` properties of the object. For example,

```
% match only ert.tlc
config.SystemTargetFile = {'ert.tlc'};
% match the standard ert TMF's
config.TemplateMakefile = {'ert_default_tmf' ...
                           'ert_unix.tmf', ...
                           'ert_vc.tmf', ...
                           'ert_vcx64.tmf', ...
                           'ert_lcc.tmf'};
% match regular 32-bit machines and Custom for e.g. ...
% 64-bit Linux
config.TargetHWDeviceType = {'Generic->32-bit x86 ...
                              compatible'
                             'Generic->Custom'};
```

See Also

`rtw.connectivity.Config`

How To

- “Verification”
- “Creating a Connectivity Configuration for a Target”

- rtwdemo_custom_pil

Purpose

Control downloading, starting and resetting executable on target hardware

Syntax

```
rtw.connectivity.Launcher(componentArgs, builder)
```

Description

Constructor	Description
Launcher	Launches an application built by a Builder object

Constructor syntax:

```
rtw.connectivity.Launcher(componentArgs, builder)
```

Launcher controls the download, start and reset of the application (e.g. PIL application) associated with a `rtw.connectivity.Builder` object. You must make a subclass and implement the `startApplication` and `stopApplication` methods.

If necessary, you can implement a destructor method that cleans up any resources (e.g., a handle to a 3rd party download tool) when this object is cleared from memory. There is significant flexibility in how the `startApplication` and `stopApplication` methods can be implemented.

See MyPIL.Launcher for an example.

For methods, see the following table.

Method	Syntax and Description
getBuilder	`builder = obj.getBuilder`
	Returns the `rtw.connectivity.Builder` object associated with this Launcher object.

Method	Syntax and Description
startApplication	`obj.startApplication`

	Abstract method that you must implemented in a subclass. Called by Simulink to start execution of the target application, created by the `rtw.connectivity.Builder` object associated with this Launcher object.
	The startApplication method must always reset the application to its initial state by ensuring that external and static (global) variables are zero initialized.
	Use the `getApplicationExecutable` method of the associated `rtw.connectivity.Builder` object to determine the application to start, e.g.,
	
	`exe = this.getBuilder.get...`
	`ApplicationExecutable`

Method	Syntax and Description
stopApplication	`obj.stopApplication`

	Abstract method that you must implemented in a subclass. Called by Simulink to stop execution of the target application, created by the `rtw.connectivity.Builder` object associated with this Launcher object.
	Use the `getApplicationExecutable` method of the associated `rtw.connectivity.Builder` object to determine the application to stop, e.g.,
	
	`exe = this.getBuilder.get...`
	`ApplicationExecutable`

getComponentArgs	`componentArgs = obj.getComponentArgs`
	Returns the `rtw.connectivity.ComponentArgs` object associated with this Launcher object.

How To

- "Verification"
- "Creating a Connectivity Configuration for a Target"
- rtwdemo_custom_pil

Purpose 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.RtIOStream-ApplicationFramework (e.g. MyPIL.TargetFramework)
exeExtension	Filename extension of an executable for the target system. The extension depends on the makefile and compiler that are called by the `MakefileBuilder`. These are defined by the template makefile specified by the source component (e.g., the referenced model under test). For an embedded target the extension may be `'.elf'`, `'.abs'`, `'.sre'`, `'.hex'`, or others. For a Windows host-based target the extension is `'.exe'`. For a UNIX® host-based target the extension is empty, `''`.

Constructor syntax:

```
rtw.connectivity.MakefileBuilder(componentArgs,
targetApplicationFramework, exeExtension)
```

MakefileBuilder controls the customizable makefile-based build process supporting the creation of custom applications (e.g. a PIL application) that interface with a Simulink component such as a referenced model (represented as a collection of binary libraries).

To build the PIL application, you must provide a template makefile that includes the target MAKEFILEBUILDER_TGT. You can use any of the standard TMF files, e.g., `ert_unix.tmf` or `ert_vc.tmf`.

See Also

`rtw.pil.RtIOStreamApplicationFramework` | `rtw.connectivity.ComponentArgs`

How To

- “Verification”
- “Creating a Connectivity Configuration for a Target”
- rtwdemo_custom_pil

rtw.connectivity.RtIOStreamHostCommunicator

Purpose

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 `rtiostream` functions.

Constructor Arguments	
componentArgs	A `rtw.connectivity.ComponentArgs` object.
launcher	A `rtw.connectivity.Launcher` object.
rtiostreamLib	An `rtiostream` shared library that implements the host side of host-target communications.

Constructor syntax:

```
rtw.connectivity.RtIOStreamHostCommunicator(componentArgs,
launcher, rtiostreamLib)
```

This class configures host-side communications with the target by loading and initializing a shared library that implements the `rtiostream` functions.

Embedded Coder provides an implementation of this shared library to support TCP/IP communications between host and target (all platforms), as well as a Windows only version for serial communications. With TCP/IP or serial, you need only supply the target-side drivers.

For other communications protocols (e.g. USB), you must supply an appropriate shared library for the host-side of the communications link as well as the target-side drivers.

To create your instance of `rtw.connectivity.RtIOStreamHostCommunicator`, you have two options:

- Instantiate `rtw.connectivity.RtIOStreamHostCommunicator` directly, providing custom arguments to supply to the `rtiostream` shared library. This is sufficient in most cases.
- Alternatively, create a subclass of `rtw.connectivity.RtIOStreamHostCommunicator`. This may be necessary when more complex configuration is required. For example, the demo subclass `rtw.connectivity.HostTCPIPCommunicator` includes additional code to determine the TCP/IP port number on which the executable application is serving, or you could use a subclass to specify a serial port number, or specify verbose or silent operation.

Methods	
setTimeoutRecvSecs	Sets the timeout value for reading data.
`hostCommunicator.setTimeoutRecvSecs(`*`timeout`*`)` configures data reading to time out if no new data is received for a period of greater than `timeout` seconds.	
setInitCommsTimeout	Sets the timeout value for initial setup of the communications channel.
`hostCommunicator.setInitCommsTimeout(`*`timeout`*`)` For some targets you may need to set a timeout value for initial setup of the communications channel. For example, the target processor may take a few seconds before it is ready to open its side of the communications channel. If you set a nonzero timeout value then the communicator repeatedly tries to open the communications channel until the timeout value is reached.	

See Also

rtw.connectivity.ComponentArgs | rtw.connectivity.Launcher | rtiostream_wrapper

How To

- "Verification"
- "Creating a Connectivity Configuration for a Target"
- rtwdemo_custom_pil

Purpose

Create and configure timer object for target

Syntax

`hw_timer_obj = rtw.connectivity.Timer`

Description

Constructor	Description
`rtw.connectivity.Timer`	Create timer object

If your hardware target does not have built-in timer support, you must create a timer object that provides details of the hardware-specific timer and any associated source files.

To create this timer object, you must create an object of type `rtw.connectivity.Timer`. For example,

```
hw_timer_obj = rtw.connectivity.Timer
```

You may use the `rtw.connectivity.Timer` class directly or make a subclass of `rtw.connectivity.Timer`. Use the following methods to configure the hardware timer object.

Method	Syntax and Description
`setTimerDataType`	*`hw_timer_obj`*`.setTimerDataType(`*`data_type`*`);`
	Specify data type. Select `'uint8'`, `'uint16'`, or `'uint32'`. For example, `TimerObj.setTimerDataType('uint32');`
`setTicksPerSecond`	*`hw_timer_obj`*`.setTicksPerSecond(`*`no_ticks_ps`*`)`

	Specify number of timer ticks per second. For example, if timer runs at 1 MHz, then number of ticks per second is 10^6.
	
	`ticksPerSecond = 1e6;`
	`TimerObj.setTicksPerSecond(ticksPerSecond);`
	
	Property is empty if you do not specify a rate.

Method	Syntax and Description
`setCountDirection`	*`hw_timer_obj`*`.setCountDirection(`*`direction`*`)`
	The default value is `'up'`, which assumes that the hardware timer increments with each clock cycle. If your hardware timer decrements with each clock cycle, set the value to `'down'`.For example, `TimerObj.setCountDirection('down');`
`setReadTimerExpression`	*`hw_timer_obj`*`.setReadTimerExpression(`*`valid_C_expression`*`)`

	Specify string to read timer. String must be a valid C expression, for example, function call `'read_timer()'`, or name of timer register that can be read directly.
	
	`readTimerExpression = 'micros()';`
	`TimerObj.setReadTimerExpression(readTimerExpression);`

`setSourceFile`	*`hw_timer_obj`*`.setSourceFile(`*`timer_source_file`*`);`

	Specify name of source file that defines timer read function. Name must include file path.Required if you are providing a custom source file that defines timer read function.
	
	For example,
	
	`timerSourceFile = fullfile(matlabroot,...`
	`'toolbox',...`
	`'rtw',...`
	`'targets',...`
	`'pil',...`
	`'host_timer_x86.c');`
	`TimerObj.setSourceFile(timerSourceFile);`

Method	Syntax and Description
`setHeaderFile`	*hw_timer_obj*.setHeaderFile(*header_file*)

	Specify name of header file that has function prototype for timer read function. Name must include file path.File may specify function prototype or define macro for accessing timer register directly.
	For example,
	`headerFile = fullfile(matlabroot,... 'toolbox',... 'rtw',... 'targets',... 'pil',... 'host_timer_x86.h');`
	`TimerObj.setHeaderFile(headerFile);`

The following listing `TimerX.m` is an example of how you create a subclass of `rtw.connectivity.Timer`:

```
classdef TimerX < rtw.connectivity.Timer
  methods
    function this = TimerX

    % Configure data type returned by timer reads
    this.setTimerDataType('uint32');

    % The micros() function returns microseconds
    ticksPerSecond = 1e6;
    this.setTicksPerSecond(ticksPerSecond);

    % The timer counts upwards
    this.setCountDirection('up');

    % Configure source files required to access the timer
    timerSourceFile = fullfile(matlabroot,...
                               'toolbox',...
```

```
                                         'rtw',...
                                         'targets',...
                                         'pil',...
                                         'host_timer_x86.c');

    headerFile = fullfile(matlabroot,...
                                    'toolbox',...
                                    'rtw',...
                                    'targets',...
                                    'pil',...
                                    'host_timer_x86.h');

    this.setSourceFile(timerSourceFile);
    this.setHeaderFile(headerFile);

    % Configure the expression used to read the timer
    readTimerExpression = 'micros()';
    this.setReadTimerExpression(readTimerExpression);
    end
  end
end
```

How To

- “Verification”
- “Creating a Connectivity Configuration for a Target”

Purpose

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

Syntax

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

Description

`obj = RTW.getEncapsulationInterfaceSpecification(modelName)` returns a handle to a model-specific C++ encapsulation interface control object.

Input Arguments

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

Output Arguments

`obj`	Handle to the C++ encapsulation interface control object associated with the specified model. If the model does not have any associated C++ encapsulation interface control object, the function returns `[]`.

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

RTW.getFunctionSpecification

Purpose

Get handle to model-specific C prototype function control object

Syntax

`obj = RTW.getFunctionSpecification(modelName)`

Description

`obj = RTW.getFunctionSpecification(modelName)` returns a handle to the model-specific C function prototype control object.

Input Arguments

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

Output Arguments

obj	Handle to the model-specific C prototype function control object associated with the specified model. If the model does not have any associated function control object, the function returns `[]`.

Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. Once you validate and apply your changes, you can generate code based on your C function prototype modifications. See "Configuring Function Prototypes" in the Embedded Coder documentation.

How To

- "Function Prototype Control"

Superclasses

ModelCPPClass

Purpose

Control C++ encapsulation interfaces for models using I/O arguments style step method

Description

The `ModelCPPArgsClass` class provides objects that describe C++ encapsulation interfaces for models using an I/O arguments style step method. Use the `attachToModel` method to attach a C++ encapsulation interface to a loaded ERT-based Simulink model.

Construction

RTW.ModelCPPArgsClass	Create C++ encapsulation interface object for configuring model class with I/O arguments style step method

Methods

See the methods of the base class `RTW.ModelCPPClass`, plus the following methods.

getArgCategory	Get argument category for Simulink model port from model-specific C++ encapsulation interface
getArgName	Get argument name for Simulink model port from model-specific C++ encapsulation interface
getArgPosition	Get argument position for Simulink model port from model-specific C++ encapsulation interface
getArgQualifier	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface

runValidation	Validate model-specific C++ encapsulation interface against Simulink model
setArgCategory	Set argument category for Simulink model port in model-specific C++ encapsulation interface
setArgName	Set argument name for Simulink model port in model-specific C++ encapsulation interface
setArgPosition	Set argument position for Simulink model port in model-specific C++ encapsulation interface
setArgQualifier	Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface

Copy Semantics

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"

- “C++ Encapsulation Interface Control”

RTW.ModelCPPArgsClass

Purpose

Create C++ encapsulation interface object for configuring model class with I/O arguments style step method

Syntax

`obj = RTW.ModelCPPArgsClass`

Description

`obj = RTW.ModelCPPArgsClass` returns a handle, `obj`, to a newly created object of class `RTW.ModelCPPArgsClass`.

Output Arguments

`obj`	Handle to a newly created C++ encapsulation interface object for configuring a model class with an I/O arguments style step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. See “Generating and Configuring C++ Encapsulation Interfaces to Model Code” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

Control C++ encapsulation interfaces for models

Description

The `ModelCPPClass` class is the base class for the classes `RTW.ModelCPPArgsClass` and `RTW.ModelCPPVoidClass`, which provide objects that describe C++ encapsulation interfaces for models using either an I/O arguments style step method or a `void-void` style step method. Use the `attachToModel` method to attach a C++ encapsulation interface to a loaded ERT-based Simulink model.

Construction

To access the methods of this class, use the constructor for either `RTW.ModelCPPArgsClass` or `RTW.ModelCPPVoidClass`.

Methods

attachToModel	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
getClassName	Get class name from model-specific C++ encapsulation interface
getDefaultConf	Get default configuration information for model-specific C++ encapsulation interface from Simulink model
getNumArgs	Get number of step method arguments from model-specific C++ encapsulation interface
getStepMethodName	Get step method name from model-specific C++ encapsulation interface

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

Alternatives

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

Superclasses ModelCPPClass

Purpose Control C++ encapsulation interfaces for models using `void-void` style step method

Description The `ModelCPPVoidClass` class provides objects that describe C++ encapsulation interfaces for models using a `void-void` style step method. Use the `attachToModel` method to attach a C++ encapsulation interface to a loaded ERT-based Simulink model.

Construction

RTW.ModelCPPVoidClass	Create C++ encapsulation interface object for configuring model class with `void-void` style step method

Methods See the methods of the base class `RTW.ModelCPPClass`, plus the following method.

runValidation	Validate model-specific C++ encapsulation interface against Simulink model

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

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

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

Purpose

Create C++ encapsulation interface object for configuring model class with `void-void` style step method

Syntax

`obj = RTW.ModelCPPVoidClass`

Description

`obj = RTW.ModelCPPVoidClass` returns a handle, `obj`, to a newly created object of class `RTW.ModelCPPVoidClass`.

Output Arguments

`obj`	Handle to a newly created C++ encapsulation interface object for configuring a model class with a `void-void` style step method. The object has not yet been configured or attached to an ERT-based Simulink model.

Alternatives

The **Configure C++ Encapsulation Interface** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Configure C++ encapsulation interface dialog box, where you can flexibly control the C++ encapsulation interfaces that are generated for your model. See "Generating and Configuring C++ Encapsulation Interfaces to Model Code" in the Embedded Coder documentation.

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

RTW.ModelSpecificCPrototype

Purpose

Describe signatures of functions for model

Description

A `ModelSpecificCPrototype` object describes the signatures of the step and initialization functions for a model. You must use this in conjunction with the `attachToModel` method.

Construction

RTW.ModelSpecificCPrototype	Create model-specific C prototype object

Methods

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

getDefaultConf	Get default configuration information for model-specific C function prototype from Simulink model
getFunctionName	Get function name from model-specific C function prototype
getNumArgs	Get number of function arguments from model-specific C function prototype
getPreview	Get model-specific C function prototype code preview
runValidation	Validate model-specific C function prototype against Simulink model
setArgCategory	Set argument category for Simulink model port in model-specific C function prototype
setArgName	Set argument name for Simulink model port in model-specific C function prototype
setArgPosition	Set argument position for Simulink model port in model-specific C function prototype
setArgQualifier	Set argument type qualifier for Simulink model port in model-specific C function prototype
setFunctionName	Set function name in model-specific C function prototype

Copy Semantics

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

Examples

The code below creates a function control object, `a`, and uses it to add argument configuration information to the model.

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

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

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

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

Alternatives

You can create a function control object using the Model Interface dialog box.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

How To

- "Function Prototype Control"

Purpose

Create model-specific C prototype object

Syntax

`obj = RTW.ModelSpecificCPrototype`

Description

`obj = RTW.ModelSpecificCPrototype` creates a handle, *obj*, to an object of class `RTW.ModelSpecificCPrototype`.

Output Arguments

obj	Handle to model specific C prototype object.

Examples

Create a function control object, `a`, and use it to add argument configuration information to the model:

```
% Open the rtwdemo_counter model and specify the System Target File
rtwdemo_counter
set_param(gcs,'SystemTargetFile','ert.tlc')

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

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

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

Alternatives

The **Configure Model Functions** button on the **Interface** pane of the Simulink Configuration Parameters dialog box launches the Model Interface dialog box, which provides you flexible control over the C function prototypes that are generated for your model. See "Configuring Function Prototypes" in the Embedded Coder documentation.

See Also

`RTW.ModelSpecificCPrototype.addArgConf`

How To

- “Function Prototype Control”

Purpose

Configure target-side communications

Syntax

```
applicationFramework = rtw.pil.RtIOStreamApplicationFramework(
   componentArgs)
```

Description

Constructor	Description
RtIOStreamApplicationFramework	Specify target-specific libraries and source files that are required to build the executable.

Constructor Argument	
componentArgs	A `rtw.connectivity.ComponentArgs` object.

Constructor syntax:

```
applicationFramework =
rtw.pil.RtIOStreamApplicationFramework(componentArgs)
```

You must create a subclass of `rtw.pil.RtIOStreamApplicationFramework`. The purpose of this class is to specify target-specific libraries and source files that are required to build the executable for the PIL application. These libraries and source files must include the device drivers that implement the target-side of the `rtiostream` communications channel. See also `rtiostream_wrapper`.

The class provides an `RTW.BuildInfo` object containing PIL-specific files (including a PIL main) that will be combined with the PIL component libraries, by the `rtw.connectivity.MakefileBuilder`, to create the PIL application. You must make a subclass and add source files, libraries, include paths and preprocessor macro definitions that are required to implement the `rtiostream` target communications interface to the `RTW.BuildInfo` object (access via `getBuildInfo` method).

The software uses only the following data in the `RTW.BuildInfo` object:

- Source file names returned by `getSourceFiles`
- Source file paths returned by `getSourcePaths`
- Include file names returned by `getIncludeFiles`
- Include file paths returned by `getIncludePaths`
- Libraries
- Preprocessor macro definitions returned by `getDefines`
- Linker options returned by `getLinkFlags`

The software ignores any other data, for example, template makefile (TMF) tokens and compiler options.

For methods that belong to `rtw.pil.RtIOStreamApplicationFramework`, see the following table.

Method	Syntax and Description
`getComponentArgs`	`componentArgs = obj.getComponentArgs`
	Returns the `rtw.connectivity.ComponentArgs` object associated with this object.
`getBuildInfo`	`buildInfo = obj.getBuildInfo`
	Returns the `RTW.BuildInfo` object associated with this object.

Method	Syntax and Description
addPILMain	`obj.addPILMain(type)`

	To build the PIL application you must specify a `main.c` file. Use the `addPILMain` method to add one of the two provided files to the application framework.
	Use the `type` argument to specify `'target'` or `'host'`, depending on which one of the following example PIL `main.c` files you want to use.
	1) To specify a `main.c` adapted for on-target PIL and suitable for most PIL implementations, enter:
	
	`` obj.addPILMain(`target') ``
	
	2) To specify a `main.c` adapted for host-based PIL, for example, as used in the `mypil` host example, enter:
	
	`` obj.addPILMain(`host') ``

See Also

`rtw.connectivity.ComponentArgs` | `rtiostream_wrapper`

How To

- “Verification”
- “Creating a Connectivity Configuration for a Target”
- “Build Information Object”
- rtwdemo_custom_pil

Purpose

Execute CGV object

Syntax

result = *cgvObj*.run()

Description

result = *cgvObj*.run() executes the model once for each input data that you added to the object. *result* is a boolean value that indicates whether the run completed without execution error. *cgvObj* is a handle to a cgv.CGV object.

After each execution of the model, the object captures and writes the following metadata to a file in the output folder:

ErrorDetails — If errors occur, the error information.
status — The execution status.
ver — Version information for MathWorks® products.
hostname — Name of computer.
dateTime — Date and time of execution.
warnings — If warnings occur, the warning messages.
username — Name of user.
runtime — The amount of time that lapsed for the execution.

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 for details.
- You can call run once without first calling cgv.CGV.addInputData. However, it is recommended that you first save all of the required data for execution to a MAT-file, including the model inputs and parameters. Then use cgv.CGV.addInputData to pass the MAT-file to the CGV object before calling run.
- 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...	cgv.CGV.run() executes callback function...
HeaderReportFcn	cgv.CGV.addHeaderReportFcn	Before executing any input data in cgv.CGV
PreExecReportFcn	cgv.CGV.addPreExecReportFcn	Before executing each input data file in cgv.CGV
PreExecFcn	cgv.CGV.addPreExecFcn	Before executing each input data file in cgv.CGV
PostExecReportFcn	cgv.CGV.addPostExecReportFcn	After executing each input data file in cgv.CGV
PostExecFcn	cgv.CGV.addPostExecFcn	After executing each input data file in cgv.CGV
TrailerReportFcn	cgv.CGV.addTrailerReportFcn	After all input data is executed in cgv.CGV

How To

- "Numerical Equivalence Checking"

RTW.AutosarInterface.runValidation

Purpose

Validate `RTW.AutosarInterface` object against model

Syntax

`[Status, Message] = autosarInterfaceObj.runValidation`

Description

`[Status, Message] = autosarInterfaceObj.runValidation` runs a validation check for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object. This check is made against the model to which *`autosarInterfaceObj`* is attached.

Before calling `runValidation`, you must call `attachToModel`.

The method `runValidation` performs the checks described in the following tables. The first table describes validation checks for all AUTOSAR use cases, and the second table describes specific validation checks when exporting multiple runnable entities.

Validation Checks

Group	Check
Valid names and paths	Runnable names and event names must all be unique, and must be valid AUTOSAR short name identifiers (see definition 1 following).
	AUTOSAR port, interface, and data element names must be valid AUTOSAR short name identifiers (see definition 1 following).
	AUTOSAR XML options for the component name, internal behavior name, and implementation name must be valid AUTOSAR path and short name identifiers (see definition 2 following).
	AUTOSAR XML options for the interface package name and data type package name must be valid AUTOSAR path identifiers (see definition 3 following).

Validation Checks (Continued)

Group	Check
Valid names and paths for sender/receiver ports	For sender/receiver ports (Implicit or explicit data access mode): • Simulink ports may have duplicated `AUTOSAR port` names, however the AUTOSAR `Interface` name must also be the same. • A Simulink inport and an outport cannot have the same `AUTOSAR port` name. • For any duplicated `AUTOSAR port` name and AUTOSAR `Interface` name, the `Data element` names must be unique. • Sender/receiver ports `AUTOSAR port` name cannot be the same as the `ServiceName` of a basic software port. • Sender/receiver ports `AUTOSAR port` name and `Interface` cannot be the same as the port name or interface of a calibration object. • Sender/receiver ports `Interface` plus XML Option `Interface package` (e.g., of the form `AUTOSAR/Service/servicename`) cannot be the same as the `ServiceInterface` of a basic software port.

Validation Checks (Continued)

Group	Check
Valid names and paths for basic software ports	For basic software ports: • `ServiceName` and `ServiceOperation` must be valid AUTOSAR short name identifiers (see definition 1 following); and `ServiceInterface` must be a valid AUTOSAR path identifier (see definition 3 following). • Simulink ports may have duplicated `ServiceName`, however the `ServiceInterface` must also be the same. • For any duplicated `ServiceName` and `ServiceInterface`, the `ServiceOperation` must be unique. • For duplicated `ServiceOperation` and `ServiceInterface`, the `ServiceName` must be unique. • Basic software port `ServiceName` name and `ServiceInterface` cannot be the same as the port name or interface of a calibration object.
Unsupported features	Model must not contain custom code blocks.
	Model must not contain continuous time.
	Model must not contain noninlined S-functions.
	Model must not contain nonfinite numbers.
	Model must not contain complex numbers.
	Model must not contain multitasking
	Model must not contain asynchronous rates
	Storage class of root I/O ports must be `auto`.
	I/O must be 1D or scalar.

Validation Checks (Continued)

Group	Check
	The sample time of a runnable must be a positive real scalar. Sample times with offset, e.g. [2 1], cause an error message.
Error status validation	An error status inport cannot point to itself (i.e., cannot specify itself as the inport for which it permits access to error status).
	Error status inports can only be defined to correspond to other inports that have Data Access Mode set to ImplicitReceive or ExplicitReceive
	Each receiver port can have only one error status port designate it as its error status.

Multiple Runnable Validation Checks

Group	Check
Wrapper subsystem validation when exporting multiple runnables. The "wrapper subsystem" is the top diagram runnables are exported from.	"Top-level" function-call subsystems (that are in the top diagram of the wrapper subsystem) must not be reusable functions. Their **Code Generation > Function Packaging** option must be set to `'Auto'`, `'Function'` or `'Inline'`.
	Top-level function-call subsystems cannot emit function calls.
	The only subsystems allowed at the top diagram are function-call subsystems, and empty subsystems (e.g., subsystems that contain no executable blocks, which may be used to display text in the model, or to double-click for help callback.)

Multiple Runnable Validation Checks (Continued)

Group	Check
	Top-level function-call subsystems cannot have wide trigger ports.
	A signal connected to an outport of the wrapper subsystem cannot have multiple destinations. The signal must have one destination that is uniquely a sender, service, or interrunnable variable.
	A signal connected to an outport of the wrapper subsystem cannot have an inport of that subsystem as its source.
	All data store memory blocks referenced from subsystems must be contained in the subsystems, to prevent data integrity issues.
	All lines must be contiguous. No line in the wrapper subsystem can be an output of a virtual Bus Creator or Mux block
	Constant blocks are not allowed in the wrapper subsystem.
	No Mux, or Demux blocks are allowed in the wrapper subsystem, because the signals being passed via the runnable I/O must be contiguous and have an address at the base of the array.

Multiple Runnable Validation Checks (Continued)

<table>
<tr><th>Group</th><th>Check</th></tr>
<tr><td>Wrapper level Merge block validation</td><td>Merge blocks have some restrictions at wrapper level:
• A merge block is only allowed in the wrapper subsystem when the merge block output is connected to a diagram outport (not another Merge block).
• The input to a Merge block in the wrapper subsystem must be connected to a function-call subsystem outport.
• The input to a Merge block in the wrapper subsystem does not need a label.
• A merge block in the wrapper subsystem cannot merge signals of unequal widths.
• You cannot connect a Merge block in the wrapper subsystem to more than one outport of any given function-call subsystem.</td></tr>
<tr><td rowspan="2">Other multiple runnable validation checks</td><td>All runnable names, event names, and interrunnable variable names must be unique. Lines representing interrunnable variables must be labelled with valid AUTOSAR short name identifiers. No goto-from pairs are allowed because then the signal label is not unique.</td></tr>
<tr><td>Interrunnable variables cannot be structs. All interrunnable variables must be scalar, noncomplex types. This is required by the AUTOSAR specification.

Signal lines that connect two top-level function-call subsystems represent interrunnable variables.</td></tr>
</table>

Multiple Runnable Validation Checks (Continued)

Group	Check
	Function-call subsystem output cannot be connected to its own input. An output of a function-call subsystem inside the wrapper subsystem cannot be connected to an input of same subsystem.
	The blocks in the top diagram of the wrapper subsystem must not have unconnected ports.
	Any top-level input that is Explicit Receive, Error Status, or Basic Software Service cannot be connected to more than one inport of any given function-call subsystem.
	The sample time of the inport associated with an error status must be the same sample time as its corresponding data port.
	Each function call subsystem being exported as a runnable entity must specify an AUTOSAR interface.

Output Arguments

`Status`
: Status flag indicating whether the configuration is valid. If valid, *`Status`* is true; otherwise, it is false.

`Message`
: If *`Status`* is false, *`Message`* explains why the configuration is invalid.

Definitions

The following are requirements for identifiers:

1. *AUTOSAR short name identifiers* must be composed of at most 32 characters, must begin with a letter, and can contain only

letters, numbers, and underscore characters. For example, `this_is_valid123`.

2. *AUTOSAR path and short name identifiers* must contain at least two path delimiter "`/`" characters, e.g., `/path/shortname`. Strings in between the path delimiters must be composed of at most 32 characters, must begin with a letter, and can contain only letters, numbers, and underscore characters.

3. *AUTOSAR path identifiers* must contain at least one path delimiter "`/`" characters, e.g., `/path`. Strings in between the path delimiters must be composed of at most 32 characters, must begin with a letter and can contain only letters, numbers, and underscore characters.

How To

- "Generating Code for AUTOSAR Software Components"

RTW.ModelCPPArgsClass.runValidation

Purpose

Validate model-specific C++ encapsulation interface against Simulink model

Syntax

`[status, msg] = runValidation(obj)`

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ encapsulation interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getEncapsulationInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification (`*modelName*`)`.

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.
msg	If *status* is false, *msg* contains a string of information describing why the configuration is invalid.

Alternatives

To validate a C++ encapsulation interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays success or failure status and an explanation of any failure. For more information, see “Configuring the Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

RTW.ModelCPPVoidClass.runValidation

Purpose

Validate model-specific C++ encapsulation interface against Simulink model

Syntax

`[status, msg] = runValidation(obj)`

Description

`[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C++ encapsulation interface against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getEncapsulationInterfaceSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.ModelCPPVoidClass` or `obj = RTW.getEncapsulationInterfaceSpecification (modelName)`.

Output Arguments

status	Boolean value; true for a valid configuration, false otherwise.
msg	If *status* is false, *msg* contains a string of information describing why the configuration is invalid.

Alternatives

To validate a C++ encapsulation interface in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. Click the **Validate** button to validate your current model step function configuration. The **Validation** pane displays success or failure status and an explanation of any failure. For more information, see “Configuring the Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

RTW.ModelSpecificCPrototype.runValidation

Purpose Validate model-specific C function prototype against Simulink model

Syntax

```
[status, msg] = runValidation(obj)
```

Description `[status, msg] = runValidation(obj)` runs a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached.

Before calling this function, you must call either `attachToModel`, to attach a function prototype to a loaded model, or `RTW.getFunctionSpecification`, to get the handle to a function prototype previously attached to a loaded model.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.

Output Arguments

status	True for a valid configuration; false otherwise.
msg	If *status* is false, *msg* contains a string explaining why the configuration is invalid.

Alternatives Click the **Validate** button in the Model Interface dialog box to run a validation check of the specified model-specific C function prototype against the ERT-based Simulink model to which it is attached. See "Model Specific C Prototypes View" in the Embedded Coder documentation.

How To

- "Function Prototype Control"

Purpose

Set argument category for Simulink model port in model-specific C++ encapsulation interface

Syntax

`setArgCategory(obj, portName, category)`

Description

`setArgCategory(obj, portName, category)` sets the category — `'Value'`, `'Pointer'`, or `'Reference'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` or `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the unqualified name of an inport or outport in your Simulink model.
category	String specifying the argument category — `'Value'`, `'Pointer'`, or `'Reference'` — to be set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, the change causes the argument to move to the first argument position when `attachToModel` or `runValidation` is called.

Alternatives

To set argument categories in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument categories that you can examine and modify. For more information, see “Configuring the Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

Set argument category for Simulink model port in model-specific C function prototype

Syntax

`setArgCategory(obj, portName, category)`

Description

`setArgCategory(obj, portName, category)` sets the category, `'Value'` or `'Pointer'`, of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification(modelName)`.
portName	String specifying the unqualified name of an inport or outport in your Simulink model.
category	String specifying the argument category, `'Value'` or `'Pointer'`, that you set for the specified Simulink model port.

Note If you change the argument category for an outport from `'Pointer'` to `'Value'`, it causes the argument to move to the first argument position when you call `RTW.ModelSpecificCPrototype.attachToModel` or `RTW.ModelSpecificCPrototype.runValidation`.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument categories. See "Model Specific C Prototypes View" in the Embedded Coder documentation.

How To

- "Function Prototype Control"

Purpose

Set argument name for Simulink model port in model-specific C++ encapsulation interface

Syntax

`setArgName(obj, portName, argName)`

Description

`setArgName(obj, portName, argName)` sets the argument name that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` or `obj = RTW.getEncapsulationInterfaceSpecification (modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
argName	String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

To set argument names in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument names that you can examine and modify. For more information, see "Configuring the Step Method for Your Model Class" in the Embedded Coder documentation.

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"

- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

Set argument name for Simulink model port in model-specific C function prototype

Syntax

`setArgName(obj, portName, argName)`

Description

`setArgName(obj, portName, argName)` sets the argument name corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype` or *obj* = `RTW.getFunctionSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
argName	String specifying the argument name to set for the specified Simulink model port. The argument must be a valid C identifier.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument names. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

RTW.ModelCPPArgsClass.setArgPosition

Purpose

Set argument position for Simulink model port in model-specific C++ encapsulation interface

Syntax

`setArgPosition(obj, portName, position)`

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface. The specified argument is then moved to the specified position, and other arguments shifted by one position accordingly.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass` or *obj* = `RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
position	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

To set argument positions in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the

Configure C++ Encapsulation Interface button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument positions that you can examine and modify. For more information, see "Configuring the Step Method for Your Model Class" in the Embedded Coder documentation.

How To

- "Configuring C++ Encapsulation Interfaces Programmatically"
- "Sample Script for Configuring the Step Method for a Model Class"
- "C++ Encapsulation Interface Control"

RTW.ModelSpecificCPrototype.setArgPosition

Purpose

Set argument position for Simulink model port in model-specific C function prototype

Syntax

`setArgPosition(obj, portName, position)`

Description

`setArgPosition(obj, portName, position)` sets the position — 1 for first, 2 for second, etc. — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype. The specified argument moves to the specified position, and other arguments shift by one position accordingly.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by `obj = RTW.ModelSpecificCPrototype` or `obj = RTW.getFunctionSpecification (modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
position	Integer specifying the argument position — 1 for first, 2 for second, etc. — to be set for the specified Simulink model port. The value must be greater than or equal to 1 and less than or equal to the number of function arguments.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument position. See “Model Specific C Prototypes View” in the Embedded Coder documentation.

How To

- “Function Prototype Control”

Purpose

Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface

Syntax

`setArgQualifier(obj, portName, qualifier)`

Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — of the argument that corresponds to a specified Simulink model inport or outport in a specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass` or `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
portName	String specifying the name of an inport or outport in your Simulink model.
qualifier	String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, `'const * const'`, or `'const &'` — to be set for the specified Simulink model port.

Alternatives

To set argument qualifiers in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display step method argument qualifiers that you can examine and modify. For more information, see “Configuring the Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”
- “C++ Encapsulation Interface Control”

Purpose

Set argument type qualifier for Simulink model port in model-specific C function prototype

Syntax

`setArgQualifier(obj, portName, qualifier)`

Description

`setArgQualifier(obj, portName, qualifier)` sets the type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — of the argument corresponding to a specified Simulink model inport or outport in a specified model-specific C function prototype.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype` or *obj* = `RTW.getFunctionSpecification (`*modelName*`)`.
portName	String specifying the name of an inport or outport in your Simulink model.
qualifier	String specifying the argument type qualifier — `'none'`, `'const'`, `'const *'`, or `'const * const'` — to be set for the specified Simulink model port.

Alternatives

Use the **Step function arguments** table in the Model Interface dialog box to specify argument qualifiers. See "Model Specific C Prototypes View" in the Embedded Coder documentation.

How To

- "Function Prototype Control"

RTW.AutosarInterface.setArxmlFilePackaging

Purpose

Set AUTOSAR XML packaging format

Syntax

`autosarInterfaceObj.setArxmlFilePackaging(arxmlPackaging))`

Description

`autosarInterfaceObj.setArxmlFilePackaging(arxmlPackaging))` sets the AUTOSAR XML packaging format in *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`arxmlPackaging`

Packaging format of AUTOSAR XML. Specify one of the following:

- `'Modular'` — XML descriptions in separate files
- `'Single file'` — XML descriptions in single file

See Also

`RTW.AutosarInterface.getArxmlFilePackaging`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Exporting AUTOSAR Software Component"

Purpose

Set class name in model-specific C++ encapsulation interface

Syntax

`setClassName(obj, clsName)`

Description

`setClassName(obj, clsName)` sets the class name in the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by `obj = RTW.ModelCPPArgsClass`, `obj = RTW.ModelCPPVoidClass`, or `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`.
clsName	String specifying a new name for the class described by the specified model-specific C++ encapsulation interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the model class name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the model class name, which you can examine and modify. In the `void-void step method` view, you can examine and modify the model class name without having to click a button. For more information, see “Configuring the Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”

- “C++ Encapsulation Interface Control”

Purpose

Set XML component name

Syntax

`autosarInterfaceObj.setComponentName(componentName)`

Description

`autosarInterfaceObj``.setComponentName(`*`componentName`*`)` sets the XML component name of *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`componentName`	XML component name for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.getComponentName`

"Generating Code for AUTOSAR Software Components" in the Embedded Coder documentation

RTW.AutosarInterface.setComponentType

Purpose

Set type of software component

Syntax

autosarInterfaceObj.setComponentType(*componentType*)

Description

autosarInterfaceObj.setComponentType(*componentType*) sets the type of the software component in *autosarInterfaceObj*, a model-specific RTW.AutosarInterface object.

Input Arguments

componentType

Type of software component. Either 'Application' or 'Sensor Actuator'.

See Also

RTW.AutosarInterface.getComponentType

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Specify XML package name for data type

Syntax

```
autosarInterfaceObj.setDataTypePackageName(dataTypePackageName
    )
```

Description

`autosarInterfaceObj``.setDataTypePackageName(`*`dataTypePackageName`*`)` specifies the name of the XML data type package for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`dataTypePackageName`

Name of data type package

See Also

`RTW.AutosarInterface.getDataTypePackageName`

How To

- "Preparing a Simulink Model for AUTOSAR Code Generation"
- "Generating AUTOSAR Code and Description Files"

arxml.importer.setDependencies

Purpose Set XML file dependencies

Syntax *importerObj*.setDependencies(*dependencies*)

Description *importerObj*.setDependencies(*dependencies*) sets the XML file dependencies associated with the arxml.importer object, *importerObj*.

Input Arguments

dependencies

Can be:

- a cell array of strings (for a list of dependencies)
- a char array (for a single dependency)
- or the empty array [] (for removing any dependency)

Note All atomic software components described in the XML file dependencies are ignored.

How To

- "Importing an AUTOSAR Software Component"

Purpose

Set type for event

Syntax

autosarInterfaceObj.setEventType(*EventName*, *EventType*)

Description

autosarInterfaceObj.setEventType(*EventName*, *EventType*) sets the event type for *EventName*, an event found in *autosarInterfaceObj*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

EventName

Name of event

EventType

Type of event, for example, TimingEvent or DataReceivedEvent

See Also

RTW.AutosarInterface.addEventConf

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

RTW.AutosarInterface.setExecutionPeriod

Purpose

Specify execution period for TimingEvent

Syntax

`autosarInterfaceObj.setExecutionPeriod(EP)`
`autosarInterfaceObj.setExecutionPeriod(EventName, EP)`

Description

autosarInterfaceObj.setExecutionPeriod(*EP*) specifies the execution period for the sole TimingEvent in a runnable.

autosarInterfaceObj.setExecutionPeriod(*EventName*, *EP*) allows you to specify the execution period for a named TimingEvent in a runnable.

autosarInterfaceObj is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`EP`

Execution period in seconds

`EventName`

Name of TimingEvent

See Also

`RTW.AutosarInterface.addEventConf` |
`RTW.AutosarInterface.getTriggerPortName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

Purpose

Set XML file name for `arxml.importer` object

Syntax

`*importerObj*.setFile(*filename*)`

Description

`*importerObj*.setFile(*filename*)` sets the name of the XML file associated with the `arxml.importer` object, `importerObj`.

Input Arguments

filename	XML file name. Only atomic software components described in this file can be imported.

How To

- "Importing an AUTOSAR Software Component"

RTW.ModelSpecificCPrototype.setFunctionName

Purpose

Set function name in model-specific C function prototype

Syntax

`setFunctionName(obj, fcnName, fcnType)`

Description

`setFunctionName(obj, fcnName, fcnType)` sets the step or initialization function name in the specified function control object.

Input Arguments

obj	Handle to a model-specific C prototype function control object previously returned by *obj* = `RTW.ModelSpecificCPrototype` or *obj* = `RTW.getFunctionSpecification(`*modelName*`)`.
fcnName	String specifying a new name for the function described by the function control object. The argument must be a valid C identifier.
fcnType	Optional. String specifying which function to name. Valid strings are `'step'` and `'init'`. If *fcnType* is not specified, sets the step function name.

Alternatives

Use the **Initialize function name** and **Step function name** fields in the Model Interface dialog box to specify function names. See "Model Specific C Prototypes View" in the Embedded Coder documentation.

How To

- "Function Prototype Control"

Purpose

Set name of XML implementation

Syntax

`autosarInterfaceObj.setImplementationName(implementationName)`

Description

`autosarInterfaceObj``.setImplementationName(`*`implementationName`*`)` specifies the name of the XML implementation for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`implementationName`

Name of XML implementation for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.getImplementationName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Exporting AUTOSAR Software Component"

RTW.AutosarInterface.setInitEventName

Purpose

Set initial event name

Syntax

`autosarInterfaceObj.setInitEventName(initEventName)`

Description

`autosarInterfaceObj``.setInitEventName(`*`initEventName`*`)` sets the initial event name for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`initEventName`	Initial event name for *`autosarInterfaceObj`*

How To

- `RTW.AutosarInterface.getInitEventName`
- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Set initial runnable name

Syntax

`autosarInterfaceObj.setInitRunnableName(initRunnableName)`

Description

autosarInterfaceObj.setInitRunnableName(*initRunnableName*) sets the initial runnable name for *autosarInterfaceObj*, a model-specific RTW.AutosarInterface object.

Input Arguments

initRunnableName	Initial runnable name for *autosarInterfaceObj*.

How To

- RTW.AutosarInterface.getInitRunnableName
- "Using the Configure AUTOSAR Interface Dialog Box"

RTW.AutosarInterface.setInterfacePackageName

Purpose

Set name of XML interface package

Syntax

`autosarInterfaceObj.setInterfacePackageName(interfacePkgName)`

Description

`autosarInterfaceObj.setInterfacePackageName(interfacePkgName)` specifies the name of the XML interface package for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`interfacePkgName`

Name of interface package for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.getInterfacePackageName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Set name of XML file for software component internal behavior

Syntax

```
autosarInterfaceObj.setInternalBehaviorName(internalBehaviorNa
   me)
```

Description

`autosarInterfaceObj.setInternalBehaviorName(internalBehaviorName)` specifies the name of the XML file with the software component internal behavior for *`autosarInterfaceObj`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`internalBehaviorName`

Name of XML file that specifies software component internal behavior for *`autosarInterfaceObj`*

See Also

`RTW.AutosarInterface.getInternalBehaviorName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Exporting AUTOSAR Software Component"

RTW.AutosarInterface.setIOAutosarPortName

Purpose

Set AUTOSAR port name

Syntax

```
autosarInterfaceObj.setIOAutosarPortName(portName,
   autosarPort)
```

Description

`autosarInterfaceObj.setIOAutosarPortName(portName,autosarPort)` updates the AUTOSAR port name in the configuration for the specified port.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

`portName`	Name of inport/outport (string)
`autosarPort`	AUTOSAR port name for *`portName`* (string).

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Set I/O data access mode

Syntax

```
autosarInterfaceObj.setIODataAccessMode(portName,
   dataAccessMode)
```

Description

autosarInterfaceObj.setIODataAccessMode(*portName*,*dataAccessMode*) sets the data access mode in the configuration for the specified port.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

portName	Name of inport/outport (string).
dataAccessMode	Data access mode (string). Can be one of the following: • ImplicitSend • ImplicitReceive • ExplicitSend • ExplicitReceive • QueuedExplicitReceived

How To

- RTW.AutosarInterface.getIODataAccessMode
- "Preparing a Simulink Model for AUTOSAR Code Generation"

RTW.AutosarInterface.setIODataElement

Purpose

Set I/O data element

Syntax

`autosarInterfaceObj.setIODataElement(portName,dataElement)`

Description

`autosarInterfaceObj.setIODataElement(portName,dataElement)` updates the name of the I/O data element in the configuration for the specified port.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

`portName`	Name of the inport/outport (string).
`dataElement`	Name of the I/O data element for *`portName`* (string).

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Set name of error status receiver port

Syntax

`autosarInterfaceObj.setIOErrorStatusReceiver(PortName,ESR)`

Description

autosarInterfaceObj.setIOErrorStatusReceiver(*PortName*,*ESR*) sets the receiver port name in the configuration for the port corresponding to *PortName* .

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

PortName

Name of inport/outport (string)

ESR

Name of receiver port for *PortName* (string)

See Also

RTW.AutosarInterface.getIOErrorStatusReceiver

How To

- "Configuring Ports for Basic Software and Error Status Receivers"

RTW.AutosarInterface.setIOInterfaceName

Purpose

Set I/O interface name

Syntax

```
autosarInterfaceObj.setIOInterfaceName(portName,
   interfaceName)
```

Description

autosarInterfaceObj.setIOInterfaceName(*portName*,*interfaceName*) updates the I/O interface name in the configuration for the specified port.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

By default the AUTOSAR port name, data element name, and interface name are the same as the Simulink port name.

Input Arguments

portName	Name of inport/outport (string).
interfaceName	Name of I/O interface for *portName* (string).

How To

- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Set port I/O service interface

Syntax

`autosarInterfaceObj.setIOServiceInterface(PortName, SI)`

Description

autosarInterfaceObj.setIOServiceInterface(*PortName*, *SI*) specifies the I/O service interface in the configuration for the port corresponding to *PortName*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

PortName

Name of the inport/outport (string)

SI

I/O service interface of *PortName* (string)

See Also

RTW.AutosarInterface.getIOServiceInterface

How To

- "Configuring Ports for Basic Software and Error Status Receivers"

RTW.AutosarInterface.setIOServiceName

Purpose

Set port I/O service name

Syntax

`autosarInterfaceObj.setIOServiceName(PortName, SN)`

Description

`autosarInterfaceObj.setIOServiceName(PortName, SN)` specifies the I/O service name in the configuration for the port corresponding to *`PortName`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`

Name of the inport/outport (string)

`SN`

Name of I/O service for *`PortName`* (string)

See Also

`RTW.AutosarInterface.getIOServiceName`

How To

- "Configuring Ports for Basic Software and Error Status Receivers"

Purpose

Set port I/O service operation

Syntax

`autosarInterfaceObj.setIOServiceOperation(PortName, SO)`

Description

`autosarInterfaceObj.setIOServiceOperation(PortName, SO)` sets the I/O service operation in the configuration for the port corresponding to *`PortName`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`PortName`

Inport/outport name (string)

`SO`

I/O service operation for *`PortName`*

See Also

`RTW.AutosarInterface.getIOServiceOperation`

How To

- “Configuring Ports for Basic Software and Error Status Receivers”

RTW.AutosarInterface.setIsServerOperation

Purpose

Indicate that server is specified

Syntax

`autosarInterfaceObj.setIsServerOperation(isServerOperation)`

Description

`autosarInterfaceObj.setIsServerOperation(isServerOperation)` sets the value of the property `'isServerOperation'` in *`autosarInterfaceObj`*.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`isServerOperation`	True or false (default). If true, indicates that a server is specified in *`autosarInterfaceObj`*.

How To

- "Configuring Client-Server Communication"

Purpose

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
`sim` or `normal` (default)	Mode of execution is normal simulation.
`sil`	Mode of execution is SIL.
`pil`	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

- “Numerical Equivalence Checking”

setNameSpace

Purpose

Set name space for C++ function entry in TFL table

Syntax

`setNameSpace(hEntry, nameSpace)`

Arguments

hEntry

Handle to a TFL function entry previously returned by one of the following:

- *hEntry* = `RTW.TflCFunctionEntry`
- *hEntry* = *MyCustomFunctionEntry*, where *MyCustomFunctionEntry* is a class derived from `RTW.TflCFunctionEntry`
- A call to the `registerCPPFunctionEntry` function

nameSpace

String specifying the name space in which the implementation function for the C++ function entry is defined.

Description

The `setNameSpace` function specifies the name space for a C++ function entry in a TFL table. During code generation, if the TFL function entry is matched, the software emits the name space in the generated function code (for example, `std::sin(tfl_cpp_U.In1)`).

If you created the function entry using *hEntry* = `RTW.TflCFunctionEntry` or *hEntry* = *MyCustomFunctionEntry* (that is, not using `registerCPPFunctionEntry`), then, before calling the `setNameSpace` function, you must enable C++ support for the function entry by calling the `enableCPP` function.

Examples

In the following example, the `setNameSpace` function is used to set the name space for the `sin` implementation function to `std`.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                              'Key',                    'sin', ...
                              'Priority',               100, ...
                              'ImplementationName',     'sin', ...
```

```
                                       'ImplementationHeaderFile', 'cmath' );
    fcn_entry.enableCPP();
    fcn_entry.setNameSpace('std');
```

See Also

enableCPP | registerCPPFunctionEntry

How To

- "Example: Mapping Math Functions to Target-Specific Implementations"
- "Creating Code Replacement Tables"
- "Code Replacement"

rtw.codegenObjectives.Objective.setObjectiveName

Purpose

Specify objective name

Syntax

`setObjectiveName(obj, objName)`

Description

`setObjectiveName(obj, objName)` specifies a name for the objective. The Configuration Set Objectives dialog box displays the name of the objective.

Input Arguments

obj	Handle to a code generation objective object previously created.
objName	Optional string that indicates the name of the objective. If you do not specify an objective name, the Configuration Set Objectives dialog box displays the objective ID for the objective name.

Examples

Name the objective Reduce RAM Example:

```
setObjectiveName(obj, 'Reduce RAM Example');
```

How To

- “Creating Custom Objectives”

Purpose

Specify folder

Syntax

```
cgvObj.setOutputDir('path')
cgvObj.setOutputDir('path', 'overwrite', 'on')
```

Description

cgvObj.setOutputDir('*path*') is an optional method that specifies a location where the object writes all output and metadata files for execution. *cgvObj* is a handle to a cgv.CGV object. *path* is the absolute or relative path to the folder. If the path does not exist, the object attempts to create the folder. If you do not call setOutputDir, the object uses the current working folder.

cgvObj.setOutputDir('*path*', 'overwrite', '*on*') includes the property and value pair to allow read-only files in the working directory to be overwritten. The default value for 'overwrite' is '*off*'.

How To

- "Numerical Equivalence Checking"

cgv.CGV.setOutputFile

Purpose Specify output data file name

Syntax `cgvObj.setOutputFile(InputIndex,OutputFile)`

Description *`cgvObj`*`.setOutputFile(`*`InputIndex`*`,`*`OutputFile`*`)` is an optional method that changes the default file name for the output data. *`cgvObj`* is a handle to a `cgv.CGV` object. *`InputIndex`* is a unique numeric identifier that specifies which output data to write to the file. The *`InputIndex`* is associated with specific input data.*`OutputFile`* is the name of the file, with or without the `.mat` extension.

How To

- "Numerical Equivalence Checking"

Purpose

Set periodic event name

Syntax

autosarInterfaceObj.setPeriodicEventName(*periodicEventName*)

Description

autosarInterfaceObj.setPeriodicEventName(*periodicEventName*) sets the name of the periodic event for *autosarInterfaceObj*, a model-specific RTW.AutosarInterface object.

Input Arguments

periodicEventName	Name of the periodic event for *autosarInterfaceObj*.

How To

- RTW.AutosarInterface.getPeriodicEventName
- "Using the Configure AUTOSAR Interface Dialog Box"

RTW.AutosarInterface.setPeriodicRunnableName

Purpose

Set periodic runnable name

Syntax

`autosarInterfaceObj.setPeriodicRunnableName(periodicRunnableName)`

Description

`autosarInterfaceObj.setPeriodicRunnableName(periodicRunnableName)` sets the name of the periodic runnable for *`autosarInterfaceObj`*, a model-specific `RTW.AutosarInterface` object.

Input Arguments

`periodicRunnableName`	Name of periodic runnable for *`autosarInterfaceObj`*.

How To

- `RTW.AutosarInterface.getPeriodicRunnableName`
- "Using the Configure AUTOSAR Interface Dialog Box"

Purpose

Register specified reserved identifiers to be associated with TFL table

Syntax

`setReservedIdentifiers(hTable, ids)`

Arguments

hTable

Handle to a TFL table previously returned by `hTable = RTW.TflTable`.

ids

Structure specifying reserved keywords to be registered in the TFL table. The structure must contain the following:

- `LibraryName` element, a string that specifies a TFL name: `'ANSI'`, `'ISO'`,`'GNU'`, or a TFL name of your choice.
- `HeaderInfos` element, a structure or cell array of structures containing
  - `HeaderName` element, a string that specifies the header file in which the identifiers are declared
  - `ReservedIds` element, a cell array of strings that specifies the names of the identifiers to be registered as reserved keywords

For example,

```
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

Description

In a TFL table, each function implementation name defined by a table entry will be registered as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

The `setReservedIdentifiers` function allows you to register up to four reserved identifier structures in a TFL table. One set of reserved identifiers can be associated with an arbitrary TFL, while the other three (if present) must be associated with ANSI®[1], ISO®[2], or GNU®[3] libraries.

For information about generating a list of reserved identifiers for the TFL that you are using to generate code, see "Simulink Coder Target Function Library Keywords" in the Simulink Coder documentation.

Examples

In the following example, `setReservedIdentifiers` is used to register four reserved identifier structures, for `'ANSI'`, `'ISO'`,`'GNU'`, and `'My Custom TFL'`, respectively.

```
hLib = RTW.TflTable;

% Create and register TFL entries here

.
.
.

% Create and register reserved identifiers
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{1}.HeaderInfos{2}.HeaderName = 'foo.h';
d{1}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{2}.LibraryName = 'ISO';
d{2}.HeaderInfos{1}.HeaderName = 'math.h';
d{2}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{2}.HeaderInfos{2}.HeaderName = 'foo.h';
d{2}.HeaderInfos{2}.ReservedIds = {'c', 'd'};
```

1. ANSI® is a registered trademark of the American National Standards Institute, Inc.
2. ISO® is a registered trademark of the International Organization for Standardization.
3. GNU® is a registered trademark of the Free Software Foundation.

```
d{3}.LibraryName = 'GNU';
d{3}.HeaderInfos{1}.HeaderName = 'math.h';
d{3}.HeaderInfos{1}.ReservedIds = {'a', 'b'};
d{3}.HeaderInfos{2}.HeaderName = 'foo.h';
d{3}.HeaderInfos{2}.ReservedIds = {'c', 'd'};

d{4}.LibraryName = 'My Custom TFL';
d{4}.HeaderInfos{1}.HeaderName = 'my_math_lib.h';
d{4}.HeaderInfos{1}.ReservedIds = {'y1', 'u1'};
d{4}.HeaderInfos{2}.HeaderName = 'my_oper_lib.h';
d{4}.HeaderInfos{2}.ReservedIds = {'foo', 'bar'};

setReservedIdentifiers(hLib, d);
```

How To

- "Adding Target Function Library Reserved Identifiers"
- "Code Replacement"

RTW.AutosarInterface.setServerInterfaceName

Purpose

Set name of server interface

Syntax

```
autosarInterfaceObj.setServerInterfaceName(ServerInterfaceName
    )
```

Description

autosarInterfaceObj.setServerInterfaceName(*ServerInterfaceName*) sets the name of the server interface specified in *autosarInterfaceObj*

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

ServerInterfaceName	Server interface name for *autosarInterfaceObj*.

How To

- "Configuring Client-Server Communication"

Purpose

Specify operation prototype

Syntax

```
autosarInterfaceObj.setServerOperationPrototype(operation_prot
    otype)
```

Description

autosarInterfaceObj.setServerOperationPrototype(*operation_prototype*) defines the server operation prototype for *autosarInterfaceObj*.

autosarInterfaceObj is a model-specific RTW.AutosarInterface object.

Input Arguments

operation_prototype

String with names of prototype and arguments:

operation_name(*dir1 datatype1 arg1*, *dir2 datatype2 arg2*, ..., *dirN datatypeN argN*, ...)

- *operation_name* — Name of operation
- *dirN* — Either IN or OUT, which indicates whether data is passed in or out of the function.
- *datatypeN* — Data type, which can be an AUTOSAR basic data type or record, Simulink data type, or array.
- *argN* — Name of the argument

Prototype and argument names must be valid AUTOSAR short-name identifiers.

How To

- "Configuring Client-Server Communication"

RTW.AutosarInterface.setServerPortName

Purpose

Set server port name

Syntax

`autosarInterfaceObj.setServerPortName(serverPortName)`

Description

`autosarInterfaceObj``.setServerPortName(`*`serverPortName`*`)` sets the server port name for the model-specific `RTW.AutosarInterface` object defined by *`autosarInterfaceObj`*.

Input Arguments

`serverPortName`	Name for server port of *`autosarInterfaceObj`*

How To

- "Configuring Client-Server Communication"

Purpose

Specify server type

Syntax

`autosarInterfaceObj.setServerType(serverType)`

Description

`autosarInterfaceObj``.setServerType(`*`serverType`*`)` specifies whether the server in *`autosarInterfaceObj`* is application software or AUTOSAR Basic Software.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`serverType`	Either `'Application software'` or `'Basic software'`

How To

- "Configuring Client-Server Communication"

RTW.ModelCPPClass.setStepMethodName

Purpose

Set step method name in model-specific C++ encapsulation interface

Syntax

`setStepMethodName(obj, fcnName)`

Description

`setStepMethodName(obj, fcnName)` sets the step method name in the specified model-specific C++ encapsulation interface.

Input Arguments

obj	Handle to a model-specific C++ encapsulation interface control object, such as a handle previously returned by *obj* = `RTW.ModelCPPArgsClass`, *obj* = `RTW.ModelCPPVoidClass`, or *obj* = `RTW.getEncapsulationInterfaceSpecification(modelName)`.
fcnName	String specifying a new name for the step method described by the specified model-specific C++ encapsulation interface. The argument must be a valid C/C++ identifier.

Alternatives

To set the step method name in the Simulink Configuration Parameters graphical user interface, go to the **Interface** pane and click the **Configure C++ Encapsulation Interface** button. This button launches the Configure C++ encapsulation interface dialog box, where you can display and configure the step method for your model class. In the `I/O arguments step method` view of this dialog box, click the **Get Default Configuration** button to display the step method name, which you can examine and modify. In the `void-void step method` view, you can examine and modify the step method name without having to click a button. For more information, see “Configuring the Step Method for Your Model Class” in the Embedded Coder documentation.

How To

- “Configuring C++ Encapsulation Interfaces Programmatically”
- “Sample Script for Configuring the Step Method for a Model Class”

- "C++ Encapsulation Interface Control"

setTflCFunctionEntryParameters

Purpose

Set specified parameters for function entry in TFL table

Syntax

`setTflCFunctionEntryParameters(`*`hEntry`*`, `*`varargin`*`)`

Arguments

`hEntry`
: Handle to a TFL function entry previously returned by *`hEntry`* `= RTW.TflCFunctionEntry` or *`hEntry = MyCustomFunctionEntry`*, where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`.

`varargin`
: Parameter/value pairs for the function entry. See varargin Parameters.

varargin Parameters

The following function entry parameters can be specified to the `setTflCFunctionEntryParameters` function using parameter/value argument pairs. For example,

```
setTflCFunctionEntryParameters(..., 'Key', 'sqrt', ...);
```

`Key`
: String specifying the name of the function to be replaced. The name must match one of the functions supported for replacement:

Math Functions

Note For detailed support information, see "Example: Mapping Math Functions to Target-Specific Implementations".

abs	acos	acosh	asin
asinh	atan	atan2	atanh
ceil	cos	cosh	exactrSqrt
exp	fix	floor	frexp
hypot	ldexp	ln	log

log10	max	min	mod/fmod
pow	rem	round	rSqrt
saturate	sign	sin	sincos
sinh	sqrt	round	tanh
Memory Utility Functions			
memcmp	memcpy	memset	memset2zero[1]
Nonfinite Support Utility Functions[2]			
getInf	getMinusInf	getNaN	isInf[3]
isNaN[3]			
Notes: [1] Some target processors provide optimized memset functions for use when performing a memory set to zero. The TFL API supports replacing memset to zero functions with more efficient target-specific functions. [2] Replacement of nonfinite functions is supported for Simulink code generation (not for Stateflow or MATLAB Coder code generation). [3] Replacement of isInf and isNaN is supported only for complex floating-point inputs.			

GenCallback

String specifying '' or 'RTW.copyFileToBuildDir'. The default is ''. If you specify 'RTW.copyFileToBuildDir', and if this function entry is matched and used, the function RTW.copyFileToBuildDir will be called after code generation to copy additional header, source, or object files that you have specified for this function entry to the build directory. For more information, see "Specifying Build Information for Code Replacements" in the Embedded Coder documentation.

Priority

Positive integer specifying the function entry's search priority, 0-100, relative to other entries of the same function name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for a function, the implementation with the higher priority will shadow the one with the lower priority.

`ImplType`
: Specifies the type of entry: `FCN_IMPL_FUNCT` for function or `FCN_IMPL_MACRO` for macro. The default is `FCN_IMPL_FUNCT`.

`ImplementationName`
: String specifying the name of the implementation function, for example, `'sqrt'`, which can match or differ from the `Key` name. The default is `''`.

`ImplementationHeaderFile`
: String specifying the name of the header file that declares the implementation function, for example, `'<math.h>'`. The default is `''`.

`ImplementationHeaderPath`
: String specifying the full path to the implementation header file. The default is `''`.

`ImplementationSourceFile`
: String specifying the name of the implementation source file. The default is `''`.

`ImplementationSourcePath`
: String specifying the full path to the implementation source file. The default is `''`.

Note To supply additional build information for the function entry, you can use TFL table entry functions `addAdditionalHeaderFile`, `addAdditionalIncludePath`, `addAdditionalLinkObj`, `addAdditionalLinkObjPath`, `addAdditionalSourceFile`, and `addAdditionalSourcePath`, and TFL table entry properties `AdditionalCompileFlags`, `AdditionalLinkFlags`, and `OtherFiles`. For more information. see "Specifying Build Information for Code Replacements".

`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. TFLs support function replacement based on computation or approximation method for the math functions rSqrt, sin, and cos. 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 any computation method
sin cos	RTW_CORDIC	Match the CORDIC approximation method
	RTW_DEFAULT	Match the default approximation method, None
	RTW_UNSPECIFIED	Match any approximation method

Description

The setTflCFunctionEntryParameters function sets specified parameters for a function entry in a TFL table.

Examples

In the following example, the setTflCFunctionEntryParameters function is used to set specified parameters for a TFL function entry for sqrt.

```
fcn_entry = RTW.TflCFunctionEntry;
fcn_entry.setTflCFunctionEntryParameters( ...
                                'Key',                      'sqrt', ...
                                'Priority',                 100, ...
                                'ImplementationName',       'sqrt', ...
                                'ImplementationHeaderFile', '<math.h>' );
```

How To

- "Example: Mapping Math Functions to Target-Specific Implementations"
- "Creating Code Replacement Tables"

- “Code Replacement”

Purpose Set specified parameters for operator entry in TFL table

Syntax `setTflCOperationEntryParameters(hEntry, varargin)`

Arguments

hEntry

Handle to a TFL table entry previously returned by one of the following class instantiations:

hEntry = RTW.TflCOperationEntry;	Supports operator replacement, described in "Example: Mapping Scalar Operators to Target-Specific Implementations" and "Mapping Nonscalar Operators to Target-Specific Implementations"
hEntry = RTW.TflCOperationEntry-Generator;	Provides relative scaling factor (RSF) fixed-point parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations", that are not available in RTW.TflCOperationEntry
hEntry = RTW.TflCOperationEntry-Generator_NetSlope;	Provides net slope parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations", that are not available in RTW.TflCOperationEntry
hEntry = RTW.TflBlasEntry-Generator;	Supports replacement of nonscalar operators with MathWorks BLAS functions, described in "Mapping Nonscalar Operators to Target-Specific Implementations"
hEntry = RTW.TflCBlasEntry-Generator;	Supports replacement of nonscalar operators with ANSI/ISO C BLAS functions, described in "Mapping Nonscalar Operators to Target-Specific Implementations"
hEntry = *MyCustomOperationEntry*; (where *MyCustomOperationEntry* is a class derived from RTW.TflCOperationEntry)	Supports operator replacement using custom TFL table entries, described in "Refining TFL Matching and Replacement Using Custom TFL Table Entries"

Note If you want to specify any of the parameters `SlopesMustBeTheSame`, `MustHaveZeroNetBias`, `RelativeScalingFactorF`, or `RelativeScalingFactorE` for your operator entry, instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator` rather than *`hEntry`* `= RTW.TflCOperationEntry`. If you want to use `NetSlopeAdjustmentFactor` and `NetFixedExponent`, instantiate your table entry using *`hEntry`* `= RTW.TflCOperationEntryGenerator_NetSlope`.

`varargin`
: Parameter/value pairs for the operator entry. See varargin Parameters.

varargin Parameters

The following operator entry parameters can be specified to the `setTflCOperationEntryParameters` function using parameter/value argument pairs. For example,

```
setTflCOperationEntryParameters(..., 'Key', 'RTW_OP_ADD', ...);
```

`Key`
: String specifying the operator to be replaced, among the operators supported for replacement:

Operator	Key
Addition (+)	RTW_OP_ADD
Subtraction (-)	RTW_OP_MINUS
Multiplication (*)	RTW_OP_MUL
Division (/)	RTW_OP_DIV
Data type conversion (cast)	RTW_OP_CAST
Shift left (<<)	RTW_OP_SL
Shift right (>>)	RTW_OP_SRA (arithmetic)[4] RTW_OP_SRL (logical)
Element-wise matrix multiplication (.*)	RTW_OP_ELEM_MUL[5]
Complex conjugation	RTW_OP_CONJUGATE
Transposition (.')	RTW_OP_TRANS
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN
Multiplication with transposition	RTW_OP_TRMUL
Multiplication with Hermitian transposition	RTW_OP_HMMUL

The default is 'RTW_OP_ADD'.

GenCallback
String specifying '' or 'RTW.copyFileToBuildDir'. The default is ''. If you specify 'RTW.copyFileToBuildDir',

4. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

5. For scalar multiplication, use RTW_OP_MUL.

and if this operator entry is matched and used, the function `RTW.copyFileToBuildDir` will be called after code generation to copy additional header, source, or object files that you have specified for this operator entry to the build directory. For more information, see "Specifying Build Information for Code Replacements" in the Embedded Coder documentation.

`Priority`
: Positive integer specifying the operator entry's search priority, 0-100, relative to other entries of the same operator name and conceptual argument list within this table. Highest priority is 0, and lowest priority is 100. The default is 100. If the table provides two implementations for an operator, the implementation with the higher priority will shadow the one with the lower priority.

`RoundingMode`
: String specifying the rounding mode supported by the implementation function: `'RTW_ROUND_FLOOR'`, `'RTW_ROUND_CEILING'`, `'RTW_ROUND_ZERO'`, `'RTW_ROUND_NEAREST'`, `'RTW_ROUND_NEAREST_ML'`, `'RTW_ROUND_SIMPLEST'`, `'RTW_ROUND_CONV'`, or `'RTW_ROUND_UNSPECIFIED'`. The default is `'RTW_ROUND_UNSPECIFIED'`.

`SaturationMode`
: String specifying the saturation mode supported by the implementation function: `'RTW_SATURATE_ON_OVERFLOW'`, `'RTW_WRAP_ON_OVERFLOW'`, or `'RTW_SATURATE_UNSPECIFIED'`. The default is `'RTW_SATURATE_UNSPECIFIED'`.

`SlopesMustBeTheSame`
: Boolean flag that, when set to `true`, indicates that TFL replacement request processing must check that the slopes on all arguments (input and output) are equal. The default is `false`.

  This parameter and `MustHaveZeroNetBias` can be used for fixed-point addition and subtraction replacement. Set both parameters to `true` to disregard specific slope and bias values and map relative slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.TflCOperationEntryGenerator` rather than `hEntry = RTW.TflCOperationEntry`.

`MustHaveZeroNetBias`
: Boolean flag that, when set to `true`, indicates that TFL replacement request processing must check that the net bias on all arguments is zero. The default is `false`.

This parameter and `SlopesMustBeTheSame` can be used for fixed-point addition and subtraction replacement. Set both parameters to `true` to disregard specific slope and bias values and map relative slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.TflCOperationEntryGenerator` rather than `hEntry = RTW.TflCOperationEntry`.

`RelativeScalingFactorF`
: Floating-point value specifying the slope adjustment factor (`F`) part of the relative scaling factor, $F2^E$, for relative scaling TFL entries. The default is `1.0`.

This parameter and `RelativeScalingFactorE` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `hEntry = RTW.TflCOperationEntryGenerator` rather than `hEntry = RTW.TflCOperationEntry`.

`RelativeScalingFactorE`
: Floating-point value specifying the fixed exponent (`E`) part of the relative scaling factor, $F2^E$, for relative scaling TFL entries. For example, `-3.0`. The default is `0`.

This parameter and `RelativeScalingFactorF` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `*hEntry* = RTW.TflCOperationEntryGenerator` rather than `*hEntry* = RTW.TflCOperationEntry`.

`isRSF`
: Boolean value specifying that the operator entry is a relative scaling factor (RSF) entry. Specify `true` if the values of `RelativeScalingFactorF` and `RelativeScalingFactorE` equal their defaults, `1.0` and `0`, but the entry nonetheless should be interpreted by the code generation process as an RSF entry.

`NetSlopeAdjustmentFactor`
: Floating-point value specifying the slope adjustment factor (`F`) part of the net slope, $F2^E$, for net slope TFL entries. The default is `1.0`.

This parameter and `NetFixedExponent` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using `*hEntry* = RTW.TflCOperationEntryGenerator_NetSlope` rather than `*hEntry* = RTW.TflCOperationEntry`.

`NetFixedExponent`
: Floating-point value specifying the fixed exponent (`E`) part of the net slope, $F2^E$, for net slope TFL entries. For example, `-3.0`. The default is `0`.

This parameter and `NetSlopeAdjustmentFactor` can be used for fixed-point multiplication and division replacement. Specify both parameters to map a range of slope and bias values to a replacement function.

To use this parameter, you must instantiate your table entry using *hEntry* = RTW.TflCOperationEntryGenerator_NetSlope rather than *hEntry* = RTW.TflCOperationEntry.

ImplementationName
: String specifying the name of the implementation function, for example, 's8_add_s8_s8'. The default is ''.

ImplementationHeaderFile
: String specifying the name of the header file that declares the implementation function, for example, 's8_add_s8_s8.h'. The default is ''.

ImplementationHeaderPath
: String specifying the full path to the implementation header file. The default is ''.

ImplementationSourceFile
: String specifying the name of the implementation source file, for example, 's8_add_s8_s8.c'. The default is ''.

ImplementationSourcePath
: String specifying the full path to the implementation source file. The default is ''.

Note To supply additional build information for the operator entry, you can use TFL table entry functions addAdditionalHeaderFile, addAdditionalIncludePath, addAdditionalLinkObj, addAdditionalLinkObjPath, addAdditionalSourceFile, and addAdditionalSourcePath, and TFL table entry properties AdditionalCompileFlags, AdditionalLinkFlags, and OtherFiles. For more information. see "Specifying Build Information for Code Replacements".

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;
        }
    }
```

Description

The `setTflCOperationEntryParameters` function sets specified parameters for an operator entry in a TFL table.

Examples

In the following example, the `setTflCOperationEntryParameters` function is used to set parameters for a TFL operator entry for `uint8` addition.

```
op_entry = RTW.TflCOperationEntry;
op_entry.setTflCOperationEntryParameters( ...
                  'Key',                      'RTW_OP_ADD', ...
                  'Priority',                 90, ...
                  'SaturationMode',           'RTW_SATURATE_UNSPECIFIED', ...
```

```
                      'RoundingMode',              'RTW_ROUND_UNSPECIFIED', ...
                      'ImplementationName',        'u8_add_u8_u8', ...
                      'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                      'ImplementationSourceFile', 'u8_add_u8_u8.c' );
```

In the following example, the `setTflCOperationEntryParameters` function is used to set parameters for a TFL operator entry for fixed-point `int16` division. The table entry specifies a relative scaling between the operator inputs and output in order to map a range of slope and bias values to a replacement function.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
                      'Key',                       'RTW_OP_DIV', ...
                      'Priority',                  90, ...
                      'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
                      'RoundingMode',              'RTW_ROUND_CEILING', ...
                      'RelativeScalingFactorF',    1.0, ...
                      'RelativeScalingFactorE',    -3.0, ...
                      'ImplementationName',        's16_div_s16_s16_rsf0p125', ...
                      'ImplementationHeaderFile', 's16_div_s16_s16_rsf0p125.h', ...
                      'ImplementationSourceFile', 's16_div_s16_s16_rsf0p125.c' );
```

In the following example, the `setTflCOperationEntryParameters` function is used to set parameters for a TFL operator entry for fixed-point `uint16` addition. The table entry specifies equal slope and zero net bias across operator inputs and output in order to map relative slope and bias values (rather than a specific slope and bias combination) to a replacement function.

```
op_entry = RTW.TflCOperationEntryGenerator;
op_entry.setTflCOperationEntryParameters( ...
                      'Key',                      'RTW_OP_ADD', ...
                      'Priority',                  90, ...
                      'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
                      'RoundingMode',              'RTW_ROUND_UNSPECIFIED', ...
                      'SlopesMustBeTheSame',       true, ...
                      'MustHaveZeroNetBias',       true, ...
```

```
'ImplementationName',       'u16_add_SameSlopeZeroBias', ...
'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c' );
```

How To

- “Example: Mapping Scalar Operators to Target-Specific Implementations”
- “Mapping Fixed-Point Operators to Target-Specific Implementations”
- “Creating Code Replacement Tables”
- “Code Replacement”

setTimerTicksPerSecond

Purpose Set number of timer ticks per second

Syntax *myExecutionProfile*.setTimerTicksPerSecond(*TimerTicksASec*)

Description *myExecutionProfile*.setTimerTicksPerSecond(*TimerTicksASec*) sets the number of timer ticks per second. Use this method if the "Creating a Connectivity Configuration for a Target" does not specify this value.

myExecutionProfile is a workspace variable generated by a SIL or PIL simulation.

Input Arguments

TimerTicksASec

Number of timer ticks per second

See Also getNumSectionProfiles | getSectionProfile | getTimerTicksPerSecond | display | report | getName | getSamplePeriod | getSampleOffset | getTicks | getTimes | getSectionNumber | getMaxTicks | getTotalTicks | getTotalSelfTicks | getNumCalls

How To

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"
- "Analyzing Code Execution Data"

Purpose

Specify Simulink inport that provides trigger data for DataReceivedEvent

Syntax

```
autosarInterfaceObj.setTriggerPortName(EventName,
   SimulinkInportName)
```

Description

`autosarInterfaceObj``.setTriggerPortName(`*`EventName`*`,` *`SimulinkInportName`*`)` specifies the inport that provides trigger data for *`EventName`*, a DataReceivedEvent.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

Input Arguments

`EventName`

Name of DataReceivedEvent

`SimulinkInportName`

Name of Simulink inport in model that provides trigger data

See Also

`RTW.AutosarInterface.addEventConf` | `RTW.AutosarInterface.getTriggerPortName`

How To

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Configuring Multiple Runnables for DataReceivedEvents"

slConfigUIGetVal

Purpose

Return current value for custom target configuration option

Syntax

`value = slConfigUIGetVal(hDlg, hSrc, 'OptionName')`

Input Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

Output Arguments

Current value of the specified option. The data type of the return value depends on the data type of the option.

Description

The `slConfigUIGetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUIGetVal` to read the current value of a specified target option.

Examples

In the following example, the `slConfigUIGetVal` function returns the value of the **Terminate function required** option on the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);
```

```
disp(['Value of IncludeMdlTerminateFcn was ', ...
      slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUISetEnabled` | `slConfigUISetVal`

How To

- “Defining and Displaying Custom Target Options”
- “Parameter Command-Line Information Summary”

slConfigUISetEnabled

Purpose

Enable or disable custom target configuration option

Syntax

```
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', true)
slConfigUISetEnabled(hDlg, hSrc, 'OptionName', false)
```

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'`*`OptionName`*`'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`true`
: Specifies that the option should be enabled.

`false`
: Specifies that the option should be disabled.

Description

The `slConfigUISetEnabled` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetEnabled` to enable or disable a specified target option.

If you use this function to disable a parameter that is represented in the Configuration Parameters dialog box, the parameter appears greyed out in the dialog context.

Examples

In the following example, the `slConfigUISetEnabled` function disables the **Terminate function required** option on the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

  slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
  slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

slConfigUIGetVal | slConfigUISetVal

How To

- “Defining and Displaying Custom Target Options”
- “Parameter Command-Line Information Summary”

slConfigUISetVal

Purpose

Set value for custom target configuration option

Syntax

`slConfigUISetVal(hDlg, hSrc, 'OptionName', OptionValue)`

Arguments

`hDlg`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`hSrc`
: Handle created in the context of a `SelectCallback` function and used by the System Target File Callback Interface functions. Pass this variable but do not set it or use it for any other purpose.

`'OptionName'`
: Quoted name of the TLC variable defined for a custom target configuration option.

`OptionValue`
: Value to be set for the specified option.

Description

The `slConfigUISetVal` function is used in the context of a user-written `SelectCallback` function, which is triggered when the custom target is selected in the System Target File Browser in the Configuration Parameters dialog box. You use `slConfigUISetVal` to set the value of a specified target option.

Examples

In the following example, the `slConfigUISetVal` function sets the value `'off'` for the **Terminate function required** option on the **Code Generation > Interface** pane of the Configuration Parameters dialog box.

```
function usertarget_selectcallback(hDlg, hSrc)

  disp(['*** Select callback triggered:', sprintf('\n'), ...
        '  Uncheck and disable "Terminate function required".']);

  disp(['Value of IncludeMdlTerminateFcn was ', ...
```

```
        slConfigUIGetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn')]);

    slConfigUISetVal(hDlg, hSrc, 'IncludeMdlTerminateFcn', 'off');
    slConfigUISetEnabled(hDlg, hSrc, 'IncludeMdlTerminateFcn', false);
```

See Also

`slConfigUIGetVal` | `slConfigUISetEnabled`

How To

- "Defining and Displaying Custom Target Options"
- "Parameter Command-Line Information Summary"

RTW.AutosarInterface.syncWithModel

Purpose

Synchronize configuration with model

Syntax

`autosarInterfaceObj``.syncWithModel`

Description

`autosarInterfaceObj``.syncWithModel` synchronizes the configuration with the model for the `RTW.AutosarInterface` class.

`autosarInterfaceObj` is a model-specific `RTW.AutosarInterface` object.

How To

- "Generating Code for AUTOSAR Software Components"

Purpose

Execute program loaded on processor

Syntax

```
IDE_Obj.run
IDE_Obj.run('runopt')
IDE_Obj.run(...,timeout)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

IDE_Obj.run runs the program file loaded on the referenced processor, returning immediately after the processor starts running. Program execution starts from the location of program counter (PC). Usually, the PC is positioned at the top of the executable file. However, if you stopped a running program with halt, the PC may be anywhere in the program. run starts the program from the PC current location.

If IDE_Obj references more the one processor, each processors calls run in sequence.

IDE_Obj.run('runopt') includes the parameter runopt that defines the action of the run method. The options for runopt are listed in the following table.

runopt string	Description
'run'	Executes the run and waits to confirm that the processor is running, and then returns to MATLAB.
'runtohalt'	Executes the run but then waits until the processor halts before returning. The halt can be the result of the PC reaching a breakpoint, or by direct interaction with the IDE, or by the normal program exit process.
'tohalt'	Waits until the running program has halted. Unlike the other options, this selection does not execute a run, it simply waits for the running program to halt.
'main'	This option resets the program and executes a run until the start of function 'main'.
'tofunc'	This option must be followed by an extra parameter *funname*, the name of the function to run to: `IDE_Obj.run('tofunc',funcname)` This executes a run from the present PC location until the start of function *funcname* is reached. If *funcname* is not along the program's normal execution path, *funcname* is not reached and the method times out.

In the 'run' and 'runtohalt' cases, a halt can be caused by a breakpoint, a direct interaction with the IDE, or by a normal program exit.

The following table shows the availability of the *runopt* options by IDE.

	CCS IDE	Eclipse IDE	MULTI IDE	VisualDSP++ IDE
'run'	Yes	Yes	Yes	Yes
'runtohalt'	Yes	Yes	Yes	Yes
'tohalt'	Yes		Yes	
'main'	Yes		Yes	
'tofunc'	Yes		Yes	

`IDE_Obj.run(...,timeout)` adds input argument `timeout`, to allow you to set the time out to a value different from the global timeout value. The *`timeout`* value specifies how long, in seconds, MATLAB waits for the processor to start executing the loaded program before returning.

Most often, the `'run'` and `'runtohalt'` options cause the processor to initiate execution, even when a timeout is reached. The timeout indicates that the confirmation was not received before the timeout period elapsed.

See Also

`halt` | `load` | `reset`

Purpose Save file

Syntax *IDE_Obj*.save(*filename*,*filetype*)

IDEs This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description Use *IDE_Obj*.save(*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	Eclipse IDE	MULTI IDE	VisualDSP++ IDE
'project'	Yes	Yes	Yes	Yes
'projectgroup'	No	No	No	Yes

Note The open method no longer supports the 'text' argument.

Examples To save all project files:

```
IDE_Obj.save('all')
```

To save the myproject project:

```
IDE_Obj.save('myproject')
```

To save the active project:

```
IDE_Obj.save([])
```

For VisualDSP++ IDE, to save all projects in the project groups:

```
IDE_Obj.save('all','projectgroup')
```

For VisualDSP++ IDE, to save the `myg.dpg` project group:

```
IDE_Obj.save('myg.dpg','projectgroup')
```

For VisualDSP++ IDE, to save the active project in the project groups:

```
IDE_Obj.save([],'projectgroup')
```

See Also

`adivdsp` | `close` | `load`

Purpose

Set active configuration build options

Syntax

```
IDE_Obj.setbuildopt(tool,ostr)
IDE_Obj.setbuildopt(file,ostr)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

Use *IDE_Obj*`.setbuildopt(`*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 correctly, first use the `getbuildopt` command to read a list of defined build tools.

If the VisualDSP++ and Code Composer Studio IDEs do not recognize the *ostr* argument, `setbuildopt` sets all switch settings to the default values for the build tool specified by `tool`.

If the MULTI IDE does not recognize the *ostr* argument, the IDE does not load the project.

Use *IDE_Obj*`.setbuildopt(`*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` | `getbuildopt`

Purpose

Program symbol table from IDE

Syntax

`s = IDE_Obj.symbol`

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

`s = IDE_Obj.symbol` 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
`s(1).name`	String reflecting the symbol entry name.
`s(1).address(1)`	Address or value of symbol entry.
`s(1).address(2)`	Memory page for the symbol entry. For TI C6xxx processors, the page is 0.

For MULTI IDE, this table shows a few possible elements of `s` and their interpretation.

s Structure Field	Contents of the Specified Field
`s(1).name`	String reflecting the symbol entry name.
`s(1).address`	Address or value of symbol entry.
`s(1).address`	Address or value of symbol entry in hex.

You can use field address in s as the address input argument to read and write.

It 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 a demo program on your processor. Then use symbol to return the entries stored in the symbol table in the processor.

```
s = IDE_Obj.symbol;
```

s contains all 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

Purpose

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. In the case of no input arguments, `ticcs` constructs the object with default values for all 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 `ticcs` 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 `ticcs` object.

Object	Property Name	Property	Default	Description
rtdx	'timeout'	Timeout	10.0 s	Specifies how long CCS waits for a response from the 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 correctly 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 Board                               Proc Processor    Processor
Num   Name                                Num  Name         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

Purpose

Set whether IDE window appears while IDE runs

Syntax

`IDE_Obj.visible(state)`

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio v3

Description

Use `IDE_Obj.visible(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 appropriate 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 visiblity status of IDE_Obj, enter:

```
IDE_Obj.isvisible

ans =
     0
```

Now, change the visibility of the IDE to 1, and check its visibility again.

```
IDE_Obj.visible(1)
IDE_Obj.isvisible

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`

Purpose

Write data to processor memory block

Syntax

```
mem=IDE_Obj.write(address,data)
mem=write(...,datatype)
mem=IDE_Obj.write(...,memorytype)
mem=IDE_Obj.write(...,timeout)
```

IDEs

This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description

`mem=`*`IDE_Obj`*`.write(`*`address`*`,`*`data`*`)` writes `data`, a collection of values, to the memory space of the DSP processor referenced by `IDE_Obj`.

The *`data`* argument is a scalar, vector, or array of values to write to the memory of the processor. The block to write begins from the DSP memory location given by the input parameter `address`.

The method writes the data starting from *`address`* without regard to type-alignment boundaries in the DSP. Conversely, the byte ordering of the data type is automatically applied.

Note You cannot write data to processor memory while the processor is running.

The *`address`* argument is a decimal or hexadecimal representation of a memory address in the processor. In all cases, the full memory address consist of two parts: the start address and the memory type. The memory type value can be explicitly defined using a numeric vector representation of the address.

Alternatively, the IDE_Obj object has a default memory type value which is applied if the memory type value is not explicitly incorporated into the passed address parameter. In DSP processors with only a single memory type, by setting the IDE_Obj object memory type value to zero it is possible to specify all addresses using the abbreviated (implied memory type) format.

You provide the *address* argument either as a numerical value that is a decimal representation of the DSP memory address, or as a string that write converts to the decimal representation of the start address. (Refer to function hex2dec in the *MATLAB Function Reference* that read uses to convert the hexadecimal string to a decimal value).

The following examples demonstrate how write uses the *address* argument.

address Parameter Value	Description
131082	Decimal address specification. The memory start address is 131082 and memory type is 0. This action is the same as specifying [131082 0].
[131082 1]	Decimal address specification. The memory start address is 131082 and memory type is 1.
'2000A'	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify *address* as cell array, in which case you can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array myaddress:

```
myaddress1 myaddress1{1} = 131072; myadddress1{2} =
'Program(PM) Memory';
```

`myaddress2 myaddress2{1} = '20000'; myadddress2{2} = 'Program(PM) Memory';`

`myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;`

`mem=write(...,`*`datatype`*`)` where the *`datatype`* argument defines the interpretation of the raw values written to DSP memory. The *`datatype`* argument specifies the data format of the raw memory image. The data is written starting from `address` without regard to data type alignment boundaries in the DSP. The byte ordering of the data type is automatically applied. The following MATLAB data types are supported.

MATLAB Data Type	Description
`double`	IEEE double-precision floating point value
`single`	IEEE single-precision floating point value
`uint8`	8-bit unsigned binary integer value
`uint16`	16-bit unsigned binary integer value
`uint32`	32-bit unsigned binary integer value
`int8`	8-bit signed two's complement integer value
`int16`	16-bit signed two's complement integer value
`int32`	32-bit signed two's complement integer value

`write` does not coerce data type alignment. Some combinations of *`address`* and *`datatype`* will be difficult for the processor to use.

`mem=`*`IDE_Obj`*`.write(...,`*`memorytype`*`)` adds an optional *`memorytype`* argument. Object `IDE_Obj` has a default memory type value 0 that `write` applies if the memory type value is not explicitly incorporated into the passed address parameter. In processors with only a single memory type, it is possible to specify all addresses using the implied memory type format by setting the value of the `IDE_Obj` memorytype property to zero.

`mem=`*`IDE_Obj`*`.write(...,`*`timeout`*`)` adds the optional *`timeout`* argument, which the number of seconds MATLAB waits for the write process to complete. If the *`timeout`* period expires before the write process returns a completion message, MATLAB throws an error and returns. Usually the process works correctly in spite of the error message.

Using write with VisualDSP++ IDE

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
`'program(pm) memory'`	0	Blackfin and SHARC
`'data(dm) memory'`	1	SHARC
`'data(dm) short word memory'`	2	SHARC
`'external data(dm) byte memory'`	3	SHARC
`'boot(prom) memory'`	4	SHARC

Examples

Example with VisualDSP++ IDE

These three syntax examples demonstrate how to use `write` in some common ways. In the first example, write an array of 16-bit integers to location [131072 1].

```
IDE_Obj.write([131072 1],int16([1:100]));
```

Now write a single-precision IEEE floating point value (32-bits) at address 2000A(Hex).

```
IDE_Obj.write('2000A',single(23.5));
```

For the third example, write a 2-D array of integers in row-major format (standard C programming format) at address 131072 (decimal).

```
mlarr = int32([1:10;101:110]);
IDE_Obj.write(131072,mlarr');
```

See Also

`hex2dec | read`

writemsg

Purpose

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. All 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 = IDE_Obj.rtdx;
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 successfully 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(IDE_Obj.rtdx,'ichan',indata);
```

The matrix `indata` is written by column to `ichan`. The preceding function syntax is equivalent to

```
writemsg(IDE_Obj.rtdx,'ichan',[1:9]);
```

See Also

`readmat` | `readmsg` | `write`

Purpose Configure your coder product to generate makefiles

Syntax

```
xmakefilesetup
```

IDEs This function supports the following IDEs:

- Analog Devices VisualDSP++
- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio v3

Description You can configure your coder product to generate and build your software using makefiles. This process can use the software build toolchains, such as compilers and linkers, associated with the preceding list of IDEs. However, the makefile build process does not use the graphical user interface of the IDE directly.

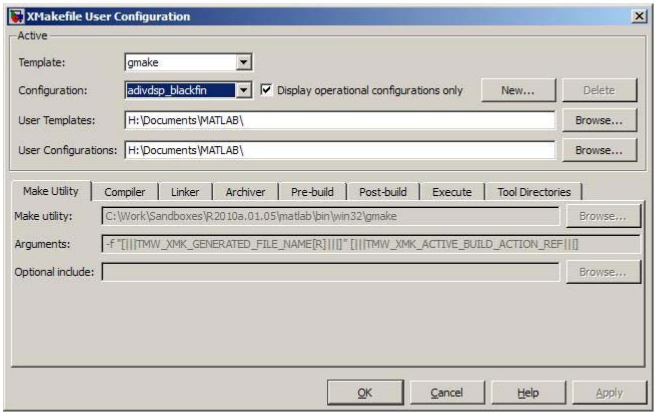
Enter `xmakefilesetup` at the MATLAB command line to configure how to generate makefiles.

Use this function:

- Before you build your software using makefiles for the first time.
- If you change the software build toolchain or processor family.

For more instructions and examples, see "Makefiles for Software Build Tool Chains".

The `xmakefile` function displays the following dialog box, which prompts you for information about your make utility and software build toolchain.



See Also

“Build format” on page 6-114 | “Build action” on page 6-116

4

Block Reference

AUTOSAR Client-Server Communication

Invoke AUTOSAR Server Operation	Configure AUTOSAR client port to access Basic Software or application software components
Mode Switch for Invoke AUTOSAR Server Operation	Toggle AUTOSAR client-server operation subsystem blocks between simulation and code generation mode

Configuration Wizards

Custom MATLAB file	Automatically update active configuration parameters of parent model using file containing custom MATLAB code
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Embedded Targets (embeddedtargetslib)

Host Communication

Byte Pack	Convert input signals to `uint8` vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP `uint8` input vector into Simulink data type values
CAN Pack	Pack individual signals into CAN message
CAN Unpack	Unpack individual signals from CAN messages
Host SCI Receive	Configure host-side serial communications interface to receive data from serial port

Host SCI Setup	Configure COM ports for host-side SCI Transmit and Receive blocks
Host SCI Transmit	Configure host-side serial communications interface to transmit data to serial port
UDP Receive	Receive UDP packet
UDP Send	Send UDP message

Target Preferences

Target Preferences	Configure model for specific IDE, tool chain, board, and processor

Embedded Linux

Linux Audio Capture	Capture ALSA audio from sound card and output data
Linux Audio Playback	Send audio data stream to ALSA audio device output
Linux Task	Spawn task function as separate Linux thread
UDP Receive	Receive UDP packet
UDP Send	Send UDP message

VxWorks

UDP Receive	Receive UDP packet
UDP Send	Send UDP message
VxWorks Task	Spawn task function as separate VxWorks thread

Analog Devices Blackfin

ADSP-BF537 EZ-KIT Lite (bf537ezkitlite)

Blackfin537 bf537_adc	Configure ADC to collect data from analog jacks and output digital data
Blackfin537 bf537_dac	Convert a stream of digital data to an analog signal and send it to the output jack
Blackfin537 bf537_uart_config	Configure UART transceiver to capture data from UART port
Blackfin537 bf537_uart_rx	Receive data stream from UART port
Blackfin537 bf537_uart_tx	Transmit data stream from UART port

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Scheduling

Blackfin Hardware Interrupt	Generate Interrupt Service Routine
Idle Task	Create free-running task

Analog Devices SHARC

Memory Operations (p. 4-7)	Memory Operations
Scheduling (p. 4-7)	Scheduling

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Scheduling

Idle Task	Create free-running task
SHARC Hardware Interrupt	Generate Interrupt Service Routine

Analog Devices TigerSHARC

Memory Operations (p. 4-7)	Memory Operations
Scheduling (p. 4-7)	Scheduling

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Scheduling

Idle Task	Create free-running task
TigerSHARC Hardware Interrupt	Generate Interrupt Service Routine

Freescale MPC55xx MPC74xx

Note Support for Freescale MPC55xx and MPC74xx processors will be removed in a future release of your MathWorks software.

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Scheduling

Idle Task	Create free-running task
MPC5500 Interrupt	Generate Interrupt Service Routine
MPC7400 Hardware Interrupt	Generate Interrupt Service Routine

Freescale MPC5xx

Note Support for Freescale MPC5xx processors will be removed in a future release of your MathWorks software.

Enhanced Queued Analog-to-Digital Converter Module-64 (p. 4-11)	Configure Queued Analog-Digital Converter (QADC64) on MPC56x (561-6) for continuous scan or digital input
Execution Profiling (p. 4-11)	Configure execution profiling over CAN or serial connection
Interrupts (p. 4-11)	Ensure data integrity between timer-based and asynchronous tasks
Modular Input/Output System (MIOS1) (p. 4-11)	Configure Modular Input/Output System (MIOS1)
Queued Analog-to-Digital Converter Module-64 (p. 4-12)	Configure Queued Analog-Digital Converter (QADC64) for continuous scan or digital input
Serial Communications Interface (SCI) (p. 4-12)	Configure serial transmit and receive
Time Processor Unit (TPU3) (p. 4-12)	Configure Time Processor Unit (TPU3)
Utilities (p. 4-13)	Configure for predefined hardware configurations

Top-Level Blocks

MPC5xx MPC555 Resource Configuration	Support device configuration for MPC5xx CPU and MIOS, QADC, and TouCAN submodules
MPC5xx Watchdog	In case of application failure, time out and reset processor

CAN 2.0B Controller Module (TouCAN)

MPC5xx CAN Calibration Protocol	Implement CAN Calibration Protocol (CCP) standard
MPC5xx TouCAN Error Count	Count transmit and receive errors detected on selected TouCAN modules
MPC5xx TouCAN Fault Confinement State	Indicate state of TouCAN module
MPC5xx TouCAN Interrupt Generator	Generate asynchronous function-call trigger when CAN interrupt occurs
MPC5xx TouCAN Receive	Receive CAN messages from TouCAN module on MPC5xx
MPC5xx TouCAN Soft Reset	Reset TouCAN module
MPC5xx TouCAN Transmit	Transmit CAN message via TouCAN module on MPC5xx
MPC5xx TouCAN Warnings	Flag excessively high transmit or receive error counts on TouCAN modules

Open CAN Message Blocks.

CAN Pack	Pack individual signals into CAN message
CAN Unpack	Unpack individual signals from CAN messages

Enhanced Queued Analog-to-Digital Converter Module-64

MPC5xx QADCE Analog In	Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode on MPC56x (561-6)
MPC5xx QADCE Digital In	Input driver enables use of Queued Analog-Digital Converter (QADC64) pins as digital inputs on MPC56x (561-566)

Execution Profiling

MPC5xx MPC555 Execution Profiling via CAN A	Provide CAN interface to execution profiling engine via CAN channel A
MPC5xx MPC555 Execution Profiling via SCI1	Provide serial interface to execution profiling engine

Interrupts

MPC5xx Asynchronous Rate Transition	Transfer data between timer-based task and asynchronous task, ensuring data integrity

Modular Input/Output System (MIOS1)

MPC5xx MIOS Digital In	Input driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM)
MPC5xx MIOS Digital Out	Output driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM)
MPC5xx MIOS Digital Out (MPWMSM)	Digital output via the MIOS Pulse Width Modulation Submodule (MPWMSM)

MPC5xx MIOS Pulse Width Modulation Out	Output driver for MIOS Pulse Width Modulation Submodule (MPWMSM)
MPC5xx MIOS Waveform Measurement	Measure pulse width and pulse period measurement via MIOS Double Action Submodule (MDASM)

Queued Analog-to-Digital Converter Module-64

MPC5xx QADC Analog In	Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode
MPC5xx QADC Digital In	Input driver enables use of Queued Analog-Digital Converter (QADC64) pins as digital inputs

Serial Communications Interface (SCI)

MPC5xx Serial Receive	Configure MPC555 for serial receive on either of QSMCM submodules SCI1 or SCI2
MPC5xx Serial Transmit	Configure MPC555 for serial transmit, using one of QSMCM submodules SCI1 or SCI2

Time Processor Unit (TPU3)

MPC5xx TPU3 Digital In	Configure Time Processor Unit (TPU3) channel for digital input
MPC5xx TPU3 Digital Out	Configure Time Processor Unit (TPU3) channel for digital output
MPC5xx TPU3 Fast Quadrature Decode	Configure pair of TPU3 channels for Fast Quadrature Decode (FQD)

MPC5xx TPU3 New Input Capture/Input Transition Counter	Configure Time Processor Unit (TPU3) channel for New Input Capture/Input Transition Counter (NITC)
MPC5xx TPU3 Programmable Time Accumulator	Configure Time Processor Unit (TPU3) channel for Programmable Time Accumulator (PTA)
MPC5xx TPU3 Pulse Width Modulation Out	Configure Time Processor Unit (TPU3) channel for pulse width modulation (PWM) output
MPC5xx TPU3 Rectangular Wave	Configure Time Processor Unit (TPU3) channel for Rectangular Wave Output (RECTW)
MPC5xx TPU3 Square Wave	Configure Time Processor Unit (TPU3) channel for Square Wave Output (SQW)

Utilities

MPC5xx Switch External Mode Configuration	Configure model for external mode or executable building
MPC5xx Switch Target Configuration	Configure model and target preferences to predefined hardware configuration

Texas Instruments C2000

C2802x (c2802xlib) (p. 4-14)	Blocks that support C2802x boards
C2803x (c2803xlib) (p. 4-15)	Blocks that support C2803x boards
C280x (c280xlib) (p. 4-17)	Blocks that support C280x boards
C281x (c281xlib) (p. 4-18)	Blocks that support C281x boards
C2834x (c2834xlib) (p. 4-19)	
C28x3x (c2833xlib) (p. 4-21)	Blocks that support C28x3x boards

Memory Operations (p. 4-22)	Memory Operations
Optimization — C28x DMC (c28xdmclib) (p. 4-22)	Blocks that represent the functionality of the TI C28x DMC Library
Optimization — C28x IQmath (tiiqmathlib) (p. 4-23)	Blocks that represent the functionality of the TI IQmath Library
RTDX Instrumentation (rtdxBlocks) (p. 4-24)	RTDX blocks for C2000 boards
Scheduling (p. 4-24)	Scheduling
Target Communication (p. 4-24)	Target Communication

C2802x (c2802xlib)

C2802x/C2803x ADC	Configure ADC to sample analog pins and output digital data
C2802x/C2803x AnalogIO Input	Configure pin, sample time, and data type for analog input
C2802x/C2803x AnalogIO Output	Configure Analog IO to output analog signals on specific pins
C2802x/C2803x COMP	Compare two input voltages on comparator pins
C280x/C2802x/C2803x/C28x3x/c2834x eCAP	Receive and log capture input pin transitions or configure auxiliary pulse width modulator
C280x/C2802x/C2803x/C28x3x/c2834x ePWM	Configure Event Manager to generate Enhanced Pulse Width Modulator (ePWM) waveforms
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input	Configure general-purpose input pins
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output	Configure general-purpose input/output pins as digital outputs

C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive	Configure inter-integrated circuit (I2C) module to receive data from I2C bus
C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit	Configure inter-integrated circuit (I2C) module to transmit data to I2C bus
C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive	Receive data on target via serial communications interface (SCI) from host
C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit	Transmit data from target via serial communications interface (SCI) to host
C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger	Generate software triggered nonmaskable interrupt
C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive	Receive data via serial peripheral interface (SPI) on target
C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit	Transmit data via serial peripheral interface (SPI) to host
C28x Watchdog	Configure counter reset source of DSP Watchdog module

C2803x (c2803xlib)

C2000 CAN Calibration Protocol	Implement CAN Calibration Protocol (CCP) standard
C2802x/C2803x ADC	Configure ADC to sample analog pins and output digital data
C2802x/C2803x AnalogIO Input	Configure pin, sample time, and data type for analog input
C2802x/C2803x AnalogIO Output	Configure Analog IO to output analog signals on specific pins
C2802x/C2803x COMP	Compare two input voltages on comparator pins

C2803x LIN Receive	Receive data via local interconnect network (LIN) module on target
C2803x LIN Transmit	Transmit data from target via serial communications interface (SCI) to host
C280x/C2802x/C2803x/C28x3x/c2834x eCAP	Receive and log capture input pin transitions or configure auxiliary pulse width modulator
C280x/C2802x/C2803x/C28x3x/c2834x ePWM	Configure Event Manager to generate Enhanced Pulse Width Modulator (ePWM) waveforms
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input	Configure general-purpose input pins
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output	Configure general-purpose input/output pins as digital outputs
C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive	Configure inter-integrated circuit (I2C) module to receive data from I2C bus
C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit	Configure inter-integrated circuit (I2C) module to transmit data to I2C bus
C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive	Receive data on target via serial communications interface (SCI) from host
C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit	Transmit data from target via serial communications interface (SCI) to host
C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger	Generate software triggered nonmaskable interrupt
C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive	Receive data via serial peripheral interface (SPI) on target
C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit	Transmit data via serial peripheral interface (SPI) to host

C280x/C2803x/C28x3x/c2834x eCAN Receive	Enhanced Control Area Network receive mailbox
C280x/C2803x/C28x3x/c2834x eCAN Transmit	Enhanced Control Area Network transmit mailbox
C280x/C2803x/C28x3x/c2834x eQEP	Quadrature encoder pulse circuit
C28x Watchdog	Configure counter reset source of DSP Watchdog module

C280x (c280xlib)

C2000 CAN Calibration Protocol	Implement CAN Calibration Protocol (CCP) standard
C280x/C2802x/C2803x/C28x3x/c2834x eCAP	Receive and log capture input pin transitions or configure auxiliary pulse width modulator
C280x/C2802x/C2803x/C28x3x/c2834x ePWM	Configure Event Manager to generate Enhanced Pulse Width Modulator (ePWM) waveforms
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input	Configure general-purpose input pins
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output	Configure general-purpose input/output pins as digital outputs
C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive	Configure inter-integrated circuit (I2C) module to receive data from I2C bus
C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit	Configure inter-integrated circuit (I2C) module to transmit data to I2C bus
C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive	Receive data on target via serial communications interface (SCI) from host

C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit	Transmit data from target via serial communications interface (SCI) to host
C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger	Generate software triggered nonmaskable interrupt
C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive	Receive data via serial peripheral interface (SPI) on target
C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit	Transmit data via serial peripheral interface (SPI) to host
C280x/C2803x/C28x3x/c2834x eCAN Receive	Enhanced Control Area Network receive mailbox
C280x/C2803x/C28x3x/c2834x eCAN Transmit	Enhanced Control Area Network transmit mailbox
C280x/C2803x/C28x3x/c2834x eQEP	Quadrature encoder pulse circuit
C280x/C28x3x ADC	Analog-to-Digital Converter (ADC)
C28x Watchdog	Configure counter reset source of DSP Watchdog module

C281x (c281xlib)

C2000 CAN Calibration Protocol	Implement CAN Calibration Protocol (CCP) standard
C281x ADC	Analog-to-digital converter (ADC)
C281x CAP	Receive and log capture input pin transitions
C281x eCAN Receive	Enhanced Control Area Network receive mailbox
C281x eCAN Transmit	Enhanced Control Area Network transmit mailbox
C281x GPIO Digital Input	General-purpose I/O pins for digital input

C281x GPIO Digital Output	General-purpose I/O pins for digital output
C281x PWM	Pulse width modulators (PWMs)
C281x QEP	Quadrature encoder pulse circuit
C281x SCI Receive	Receive data on target via serial communications interface (SCI) from host
C281x SCI Transmit	Transmit data from target via serial communications interface (SCI) to host
C281x Software Interrupt Trigger	Generate software triggered nonmaskable interrupt
C281x SPI Receive	Receive data via serial peripheral interface on target
C281x SPI Transmit	Transmit data via serial peripheral interface (SPI) to host
C281x Timer	Configure general-purpose timer in Event Manager module
C28x Watchdog	Configure counter reset source of DSP Watchdog module

C2834x (c2834xlib)

C2000 CAN Calibration Protocol	Implement CAN Calibration Protocol (CCP) standard
C280x/C2802x/C2803x/C28x3x/c2834x eCAP	Receive and log capture input pin transitions or configure auxiliary pulse width modulator
C280x/C2802x/C2803x/C28x3x/c2834x ePWM	Configure Event Manager to generate Enhanced Pulse Width Modulator (ePWM) waveforms
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input	Configure general-purpose input pins

C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output	Configure general-purpose input/output pins as digital outputs
C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive	Configure inter-integrated circuit (I2C) module to receive data from I2C bus
C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit	Configure inter-integrated circuit (I2C) module to transmit data to I2C bus
C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive	Receive data on target via serial communications interface (SCI) from host
C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit	Transmit data from target via serial communications interface (SCI) to host
C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger	Generate software triggered nonmaskable interrupt
C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive	Receive data via serial peripheral interface (SPI) on target
C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit	Transmit data via serial peripheral interface (SPI) to host
C280x/C2803x/C28x3x/c2834x eCAN Receive	Enhanced Control Area Network receive mailbox
C280x/C2803x/C28x3x/c2834x eCAN Transmit	Enhanced Control Area Network transmit mailbox
C280x/C2803x/C28x3x/c2834x eQEP	Quadrature encoder pulse circuit

C28x3x (c2833xlib)

C2000 CAN Calibration Protocol	Implement CAN Calibration Protocol (CCP) standard
C280x/C2802x/C2803x/C28x3x/c2834x eCAP	Receive and log capture input pin transitions or configure auxiliary pulse width modulator
C280x/C2802x/C2803x/C28x3x/c2834x ePWM	Configure Event Manager to generate Enhanced Pulse Width Modulator (ePWM) waveforms
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input	Configure general-purpose input pins
C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output	Configure general-purpose input/output pins as digital outputs
C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive	Configure inter-integrated circuit (I2C) module to receive data from I2C bus
C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit	Configure inter-integrated circuit (I2C) module to transmit data to I2C bus
C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive	Receive data on target via serial communications interface (SCI) from host
C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit	Transmit data from target via serial communications interface (SCI) to host
C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger	Generate software triggered nonmaskable interrupt
C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive	Receive data via serial peripheral interface (SPI) on target
C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit	Transmit data via serial peripheral interface (SPI) to host

C280x/C2803x/C28x3x/c2834x eCAN Receive	Enhanced Control Area Network receive mailbox
C280x/C2803x/C28x3x/c2834x eCAN Transmit	Enhanced Control Area Network transmit mailbox
C280x/C2803x/C28x3x/c2834x eQEP	Quadrature encoder pulse circuit
C280x/C28x3x ADC	Analog-to-Digital Converter (ADC)
C28x Watchdog	Configure counter reset source of DSP Watchdog module

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Optimization — C28x DMC (c28xdmclib)

C2000 Clarke Transformation	Convert balanced three-phase quantities to balanced two-phase quadrature quantities
C2000 Inverse Park Transformation	Convert rotating reference frame vectors to two-phase stationary reference frame
C2000 Park Transformation	Convert two-phase stationary system vectors to rotating system vectors
C2000 PID Controller	Digital PID controller
C2000 Ramp Control	Create ramp-up and ramp-down function
C2000 Ramp Generator	Generate ramp output
C2000 Space Vector Generator	Duty ratios for stator reference voltage
C2000 Speed Measurement	Calculate motor speed

Optimization — C28x IQmath (tiiqmathlib)

C2000 Absolute IQN	Absolute value
C2000 Arctangent IQN	Four-quadrant arc tangent
C2000 Division IQN	Divide IQ numbers
C2000 Float to IQN	Convert floating-point number to IQ number
C2000 Fractional part IQN	Fractional part of IQ number
C2000 Fractional part IQN x int32	Fractional part of result of multiplying IQ number and long integer
C2000 Integer part IQN	Integer part of IQ number
C2000 Integer part IQN x int32	Integer part of result of multiplying IQ number and long integer
C2000 IQN to Float	Convert IQ number to floating-point number
C2000 IQN x int32	Multiply IQ number with long integer
C2000 IQN x IQN	Multiply IQ numbers with same Q format
C2000 IQN1 to IQN2	Convert IQ number to different Q format
C2000 IQN1 x IQN2	Multiply IQ numbers with different Q formats
C2000 Magnitude IQN	Magnitude of two orthogonal IQ numbers
C2000 Saturate IQN	Saturate IQ number
C2000 Square Root IQN	Square root or inverse square root of IQ number
C2000 Trig Fcn IQN	Sine, cosine, or arc tangent of IQ number

RTDX Instrumentation (rtdxBlocks)

C2000 From RTDX	Add RTDX communication channel for target to receive data from host
C2000 To RTDX	Add RTDX communication channel to send data from target to host

Scheduling

C280x/C2802x/C2803x/C28x3x Hardware Interrupt	Interrupt Service Routine to handle hardware interrupt on C280x/C28x3x processors
C281x Hardware Interrupt	Interrupt Service Routine to handle hardware interrupt
Idle Task	Create free-running task

Target Communication

Byte Pack	Convert input signals to `uint8` vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP `uint8` input vector into Simulink data type values
CAN Pack	Pack individual signals into CAN message
CAN Unpack	Unpack individual signals from CAN messages

Texas Instruments C5000

C5510 DSK (c5510dsk)

C5510 DSK ADC	Configure AIC23 and peripherals to collect data from analog jacks and output digital data
C5510 DSK DAC	Configure AIC23 codec and peripherals to send data stream to output jack

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Scheduling

C5000/C6000 Hardware Interrupt	Interrupt Service Routine to handle hardware interrupt on C5000 and C6000 processors
Idle Task	Create free-running task

Texas Instruments C6000

AVNET S3ADSP DM6437 (avnet_s3adsp_dm6437)

C6000 Deinterleave	Separate interleaved YCbCr 4:2:2 data into Y, Cb, and Cr components
C6000 Interleave	Convert planar YCbCr 4:2:2 data to interleaved YCbCr 4:2:2 data
C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
DM643x CAN Receive	Receive messages from CAN serial communications bus on DM643x
DM643x CAN Setup	Configure CAN serial communications bus parameters on DM643x
DM643x CAN Transmit	Configure CAN mailbox to transmit messages on CAN serial communications bus on DM643x
DM643x Draw Rectangles	Configure Video Processing Back End to draw rectangles using On Screen Display (OSD) module
DM643x OSD	Overlay graphics and text on video
DM643x PWM	Configure DM643x DSP Event Manager to generate PWM waveforms
DM643x UART Config	Configure DM643x UART for serial communication
DM643x UART Receive	Configure receiver element of DM643x UART module for serial communication
DM643x UART Transmit	Configure transmitter element of DM643x UART module for serial communication

DM643x Video Capture	Configure Video Processing Front End (VPFE) to capture REC656 or generic YCbCr 4:2:2 video
DM643x Video Display	Configure Video Processing Back End to display NTSC/PAL video

C6416 DSK (c6416dsklib)

C6416 DSK ADC	Digitized output from codec to processor
C6416 DSK DAC	Use codec to convert digital input to analog output
C6416 DSK DIP Switch	Simulate or read DIP switches
C6416 DSK LED	Control LEDs
C6416 DSK Reset	Reset to initial conditions

C6455 EVM (c6455evmlib)

C6455 DSK/EVM ADC	Configure AIC23 audio codec to capture audio stream from LINE-IN or MIC
C6455 DSK/EVM DAC	Configure AIC23 codec to convert digital signal to audio output on LINE OUT and HP OUT
C6455 DSK/EVM DIP	Output state of user-selected DIP switch as Boolean
C6455 DSK/EVM LED	Apply Boolean input to user-selected LED
C6455 SRIO Config	Configure generated code for serial RapidI/O peripheral

C6455 SRIO Receive	Configure generated code to receive serial RapidI/O packets
C6455 SRIO Transmit	Configure generated code to transmit serial RapidI/O packets

C6713 DSK (c6713dsklib)

C6713 DSK ADC	Digitized signal output from codec to processor
C6713 DSK DAC	Configure codec to convert digital input to analog output
C6713 DSK DIP Switch	Simulate or read DIP switches
C6713 DSK LED	Control LEDs
C6713 DSK Reset	Reset to initial conditions

C6747 EVM (c6747evmlib)

C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
C6747 EVM DIP Switch	Output DIP switch status
C6747 EVM LED	Control four on-board LEDs
C6747 EVM/C6748 EVM ADC	Capture audio stream from LINE IN jack
C6747 EVM/C6748 EVM DAC	Output audio on LINE OUT / HP OUT jacks

DM642 EVM (dm642evmlib)

DM642 EVM Audio ADC	Audio codec and peripherals
DM642 EVM Audio DAC	Configure codec to convert digital audio input to analog audio output
DM642 EVM FPGA GPIO Read	User GPIO registers to read from selected pins
DM642 EVM FPGA GPIO Write	Write to GPIO registers
DM642 EVM LED	Control LEDs
DM642 EVM Reset	Reset to initial conditions
DM642 EVM Video ADC	Video decoders to capture analog video
DM642 EVM Video DAC	Video encoder to display video
DM642 EVM Video Port	Video port to receive video data from video input port

DM6437 EVM (dm6437evmlib)

C6000 Deinterleave	Separate interleaved YCbCr 4:2:2 data into Y, Cb, and Cr components
C6000 Interleave	Convert planar YCbCr 4:2:2 data to interleaved YCbCr 4:2:2 data
C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
DM6437 EVM ADC	Configure AIC33 audio codec to capture audio stream from LINE-IN or MIC
DM6437 EVM DAC	Configure AIC33 codec to convert digital signal to audio output on LINE OUT and HP OUT
DM6437 EVM DIP	Output state of user-selected DIP switch as Boolean

DM6437 EVM LED	Apply Boolean input to user-selected LED
DM6437 EVM Video Capture	Configure video peripherals to capture NTSC/PAL video
DM643x CAN Receive	Receive messages from CAN serial communications bus on DM643x
DM643x CAN Setup	Configure CAN serial communications bus parameters on DM643x
DM643x CAN Transmit	Configure CAN mailbox to transmit messages on CAN serial communications bus on DM643x
DM643x Draw Rectangles	Configure Video Processing Back End to draw rectangles using On Screen Display (OSD) module
DM643x OSD	Overlay graphics and text on video
DM643x PWM	Configure DM643x DSP Event Manager to generate PWM waveforms
DM643x UART Config	Configure DM643x UART for serial communication
DM643x UART Receive	Configure receiver element of DM643x UART module for serial communication
DM643x UART Transmit	Configure transmitter element of DM643x UART module for serial communication
DM643x Video Display	Configure Video Processing Back End to display NTSC/PAL video

DM648 EVM (dm648evmlib)

C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
DM648 EVM Video Capture	Configure DSP peripherals to capture NTSC/PAL or HD video
DM648 EVM Video Display	Configure DSP peripherals to display NTSC, PAL, HD, or VESA video

DSP/BIOS (dspbioslib)

DSP/BIOS Hardware Interrupt	Generate Interrupt Service Routine
DSP/BIOS Task	Create task that runs as separate DSP/BIOS thread
DSP/BIOS Triggered Task	Create asynchronously triggered task

Memory Operations

Memory Allocate	Allocate memory section
Memory Copy	Copy to and from memory section

Optimization — C62x DSP Library (tic62dsplib)

C62x Convert Floating-Point to Q.15	Convert single-precision floating-point input signal to Q.15 fixed-point
C62x Convert Q.15 to Floating-Point	Convert Q.15 fixed-point signal to single-precision floating-point

C62x Complex FIR	Filter complex input signal using complex FIR filter
C62x General Real FIR	Filter real input signal using real FIR filter
C62x LMS Adaptive FIR	LMS adaptive FIR filtering
C62x Radix-4 Real FIR	Filter real input signal using real FIR filter
C62x Radix-8 Real FIR	Filter real input signal using real FIR filter
C62x Real Forward Lattice All-Pole IIR	Filter real input signal using lattice filter
C62x Real IIR	Filter real input signal using IIR filter
C62x Symmetric Real FIR	Filter real input signal using FIR filter

C62x Autocorrelation	Autocorrelate input vector or frame-based matrix
C62x Block Exponent	Minimum number of extra sign bits in each input channel
C62x Matrix Multiply	Matrix multiply two input signals
C62x Matrix Transpose	Matrix transpose input signal
C62x Reciprocal	Fraction and exponent portions of reciprocal of real input signal
C62x Vector Dot Product	Vector dot product of real input signals
C62x Vector Maximum Index	Zero-based index of maximum value element in each input signal channel
C62x Vector Maximum Value	Maximum value for each input signal channel
C62x Vector Minimum Value	Minimum value for each input signal channel

C62x Vector Multiply	Element-wise multiplication on inputs
C62x Vector Negate	Negate each input signal element
C62x Vector Sum of Squares	Sum of squares over each real input channel
C62x Weighted Vector Sum	Weighted sum of input vectors
C62x Bit Reverse	Bit-reverse elements of each complex input signal channel
C62x FFT	
C62x Radix-2 FFT	Radix-2 decimation-in-frequency forward FFT of complex input vector
C62x Radix-2 IFFT	Radix-2 inverse FFT of complex input vector

Optimization — C64x DSP Library (tic64dsplib)

C64x Convert Floating-Point to Q.15	Convert floating-point signal to Q.15 fixed-point
C64x Convert Q.15 to Floating-Point	Convert Q.15 fixed-point signal to single-precision floating-point
C64x Complex FIR	Filter complex input signal using complex FIR filter
C64x General Real FIR	Filter real input signal using real FIR filter
C64x LMS Adaptive FIR	LMS adaptive FIR filtering
C64x Radix-4 Real FIR	Filter real input signal using real FIR filter
C64x Radix-8 Real FIR	Filter real input signal using real FIR filter

C64x Real Forward Lattice All-Pole IIR	Filter real input signal using lattice IIR filter
C64x Real IIR	Filter real input signal using IIR filter
C64x Symmetric Real FIR	Filter real input signal using FIR filter
C64x Autocorrelation	Autocorrelate input vector or frame-based matrix
C64x Block Exponent	Minimum number of extra sign bits in each input channel
C64x Matrix Multiply	Matrix multiply two input signals
C64x Matrix Transpose	Matrix transpose input signal
C64x Reciprocal	Fraction and exponent of reciprocal of real input signal
C64x Vector Dot Product	Vector dot product of real input signals
C64x Vector Maximum Index	Zero-based index of maximum value element in each input signal channel
C64x Vector Maximum Value	Maximum value for each input signal channel
C64x Vector Minimum Value	Minimum value for each input signal channel
C64x Vector Multiply	Element-wise multiplication on inputs
C64x Vector Negate	Negate each input signal element
C64x Vector Sum of Squares	Sum of squares over each real input channel
C64x Weighted Vector Sum	Weighted sum of input vectors

C64x Bit Reverse	Bit-reverse elements of each complex input signal channel
C64x FFT	Decimation-in-frequency forward FFT of complex input vector
C64x Radix-2 FFT	Radix-2 decimation-in-frequency forward FFT of complex input vector
C64x Radix-2 IFFT	Radix-2 inverse FFT of complex input vector

Scheduling

C6000 Block Processing	Repeat user-specified operation on submatrices of input matrix, using internal memory of DSP for increased efficiency
C6000 CPU Timer	Select timer and configure periodic interrupt
C6000 EDMA	Configure EDMA Controller on C6000 processor
C5000/C6000 Hardware Interrupt	Interrupt Service Routine to handle hardware interrupt on C5000 and C6000 processors
Idle Task	Create free-running task

Target Communication (targetcommlib)

Byte Pack	Convert input signals to `uint8` vector
Byte Reversal	Reverse order of bytes in input word
Byte Unpack	Unpack UDP `uint8` input vector into Simulink data type values

C6000 IP Config	Configure Internet Protocol on C6000 targets with Ethernet ports
C6000 TCP/IP Receive	Receive message from remote IP interface
C6000 TCP/IP Send	Send message to remote IP interface
C6000 UDP Receive	Receive `uint8` vector as UDP message
C6000 UDP Send	Send UDP message to host

Module Packaging

Data Object Wizard	Simulink data object wizard for creating potential Simulink data objects

Blocks — Alphabetical List

Blackfin537 bf537_adc

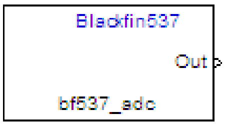
Purpose

Configure ADC to collect data from analog jacks and output digital data

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ ADSP-BF537 EZ-KIT Lite

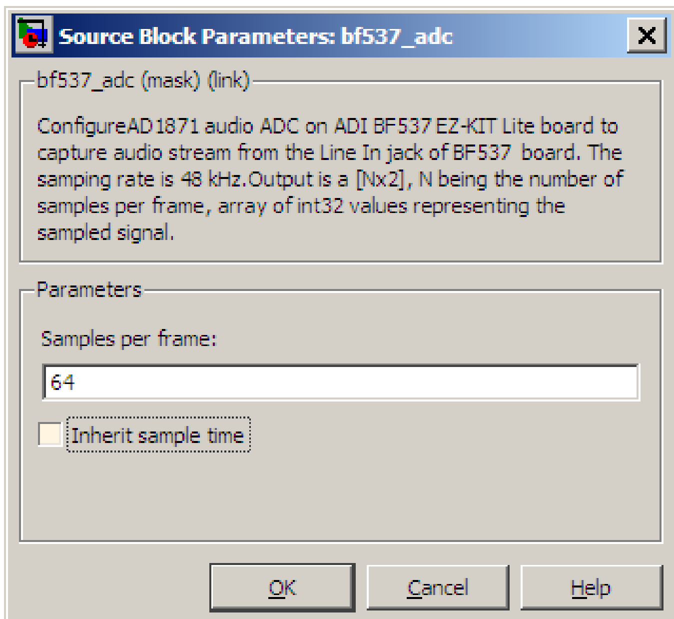
Description



bf537_adc

Configure AD1871 audio ADC on ADI BF537 EZ-KIT Lite board to capture audio stream from the Line In jack of BF537 board. This block uses a sampling rate of 48 kHz. It outputs the sampled signal as [Nx2], where N indicates number of samples per frame in an array of int32 values.

Dialog Box



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. The sample rate of the ADI BF537 EZ-KIT Lite board is 48 kHz. If you set **Samples per frame** to 64, the resulting frame rate is 750 frames per second (48000/64 = 750).

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 Interrupt, Task, or Triggered Task blocks.

References

ADSP-BF537 EZ-KIT Lite® Evaluation System Manual, Part Number 82-000865-01, available from the Analog Devices Web site.

See Also

Blackfin537 bf537_dac

Blackfin537 bf537_dac

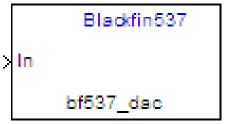
Purpose

Convert a stream of digital data to an analog signal and send it to the output jack

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ ADSP-BF537 EZ-KIT Lite

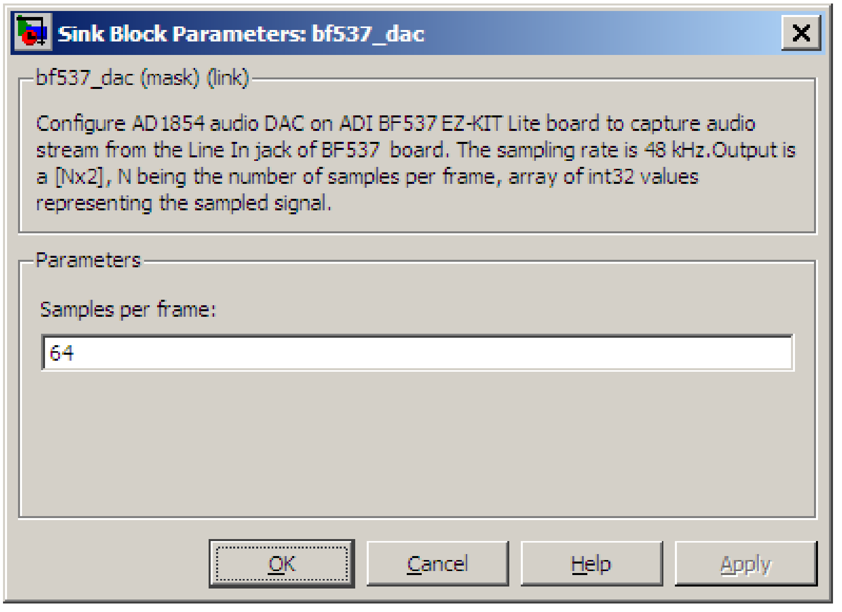
Description



bf537_dac

Configure AD1854 audio DAC on ADI BF537 EZ-KIT Lite board to capture audio stream from the Line In jack of BF537 board. This block uses a sampling rate of 48 kHz. It outputs the sampled signal as [Nx2], where N indicates number of samples per frame in an array of int32 values.

Dialog Box



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.

References

ADSP-BF537 EZ-KIT Lite® Evaluation System Manual, Part Number 82-000865-01, available from the Analog Devices Web site.

See Also

Blackfin537 bf537_adc

Blackfin537 bf537_uart_config

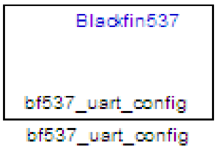
Purpose

Configure UART transceiver to capture data from UART port

Library

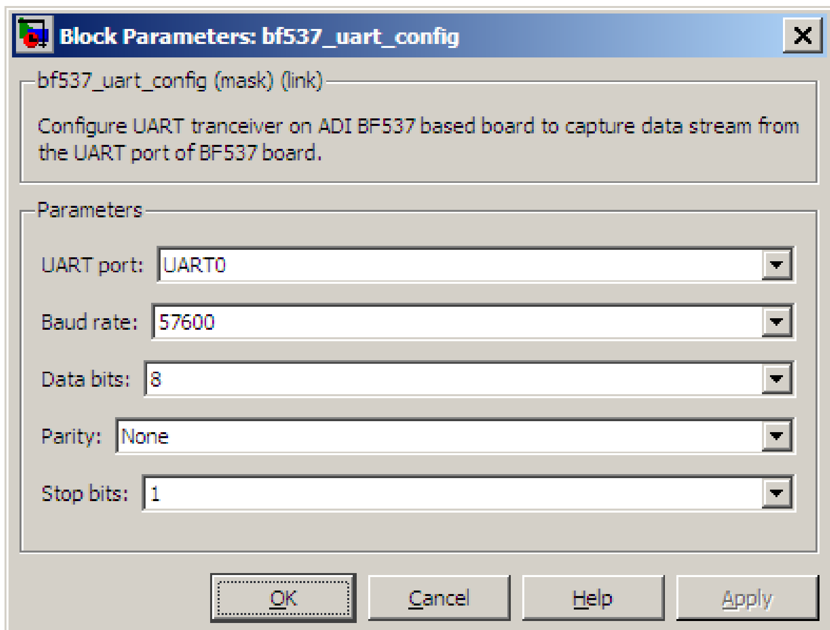
Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ ADSP-BF537 EZ-KIT Lite

Description



Configure UART transceiver on ADI BF537 based board to capture data stream from the UART port of BF537 board. Your model can only contain one configuration block per UART port.

Dialog Box



UART port

Select which UART port this block configures. UART0 uses processor pins PF0 (UART0 transmit) and PF1 (UART0 receive).

UART1 uses processor pins PF2 (Push button SW13) and PF3 (Push button SW12). These pins have multiple GPIO functions that depend on the configuration of the processor. For more information, see the “Programmable Flags (PFs)” section of the *ADSP-BF537 EZ-KIT Lite® Evaluation System Manual*.

Baud rate
Configure the rate at which the UART transfers bits per second. The bits include the start bit, the data bits, the parity bit (if enabled), and the stop bits. Configure both the sending and receiving devices to the same baud rate.

Data bits
Set the number of data bits per data frame to 5, 6, 7, or 8. The UART transmits the least significant bit sent first. Use the default value, 8 bits, unless your system requires a lower value. Configure both the sending and receiving devices to the same data bit value.

Parity
Set type of parity checking to be `none`, `even`, or `odd`. When you set **Parity** to `none`, the UART does not perform parity checking and does not transmit a parity bit. When you set **Parity** to `even`, the UART sets the parity bit to 1 to obtain an even number of ones in the data word. When you set **Parity** to `odd`, the UART sets the parity bit to 1 to obtain an odd number of ones in the data word. Parity checking can detect errors of 1 bit only. An error in 2 bits can cause the data to have a seemingly valid parity. Configure both the sending and receiving devices to the same parity value.

Stop bits
Set the number of bits used to indicate the end of a byte. When you set **Stop bits** to `1`, the UART transmits 1 bit to signal the end of a transmission. When you set **Stop bits** to `1.5`, the UART extends the length of time it transmits the 1-bit stop bit by half. Configure both the sending and receiving devices to the same stop bit value.

References

ADSP-BF537 EZ-KIT Lite® Evaluation System Manual, Part Number 82-000865-01, available from the Analog Devices Web site.

See Also Blackfin537 bf537_uart_rx, Blackfin537 bf537_uart_tx

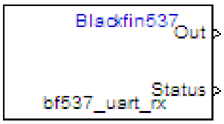
Purpose

Receive data stream from UART port

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ ADSP-BF537 EZ-KIT Lite

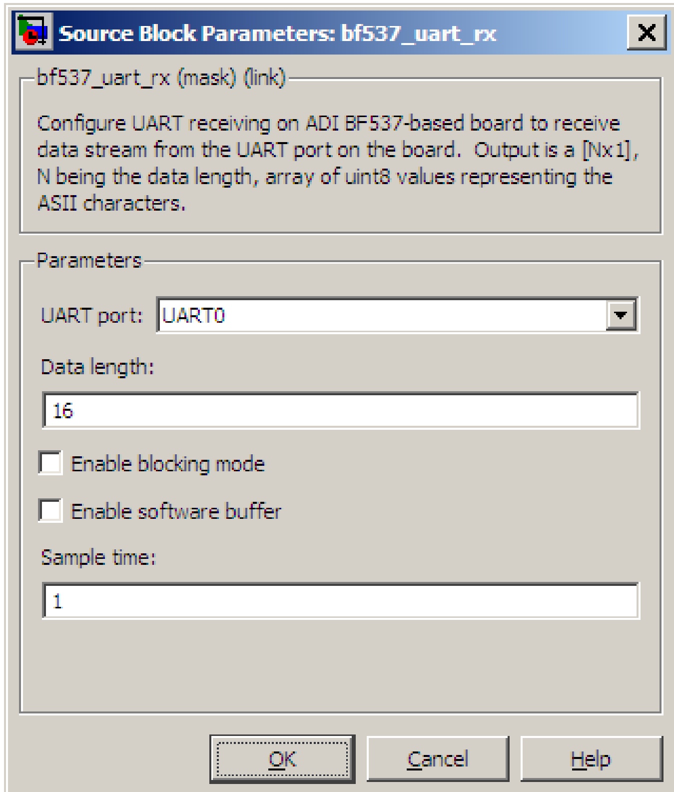
Description



bf537_uart_rx

Configure UART receiving on ADI BF537-based board to receive data stream from the UART port on the board. This block outputs [Nx1], where N indicates the data length in an array of uint8 values representing the ASCII characters. Your model can only contain one receive block per UART port.

Dialog Box



UART port

Select which UART port from which this block receives data.

Data length

Set the data length, in bytes, of the **Out** port. This block always outputs the number of bytes the **Data length** parameter specifies.

Enable blocking mode

When you enable blocking mode, this block waits until it receives enough data before writing the data to the **Out** port.

When you disable blocking mode:

- If the receive buffer contains the number of bytes specified by **Data length**, the block writes the data to the **Out** port and also sends a positive number on the **Status** port. This positive number indicates valid data on the **Out** port.
- If the receive buffer does not contain the number of bytes specified by **Data length**, the block does not write the data to the **Out** port and instead sends a 0 to the **Status** port. This 0 indicates invalid data on the out port.

Enable software buffer
Use a software-managed buffer, in addition to hardware FIFO, to handle incoming data.

Software buffer size factor
If you enable the software buffer, set the size of **Software buffer size factor** to handle expected bursts in the incoming data.

Sample time
Specify the time interval between samples. To inherit sample time from the upstream block, set this parameter to -1.

References

ADSP-BF537 EZ-KIT Lite® Evaluation System Manual, Part Number 82-000865-01, available from the Analog Devices Web site.

See Also

Blackfin537 bf537_uart_config, Blackfin537 bf537_uart_tx

Blackfin537 bf537_uart_tx

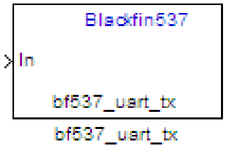
Purpose

Transmit data stream from UART port

Library

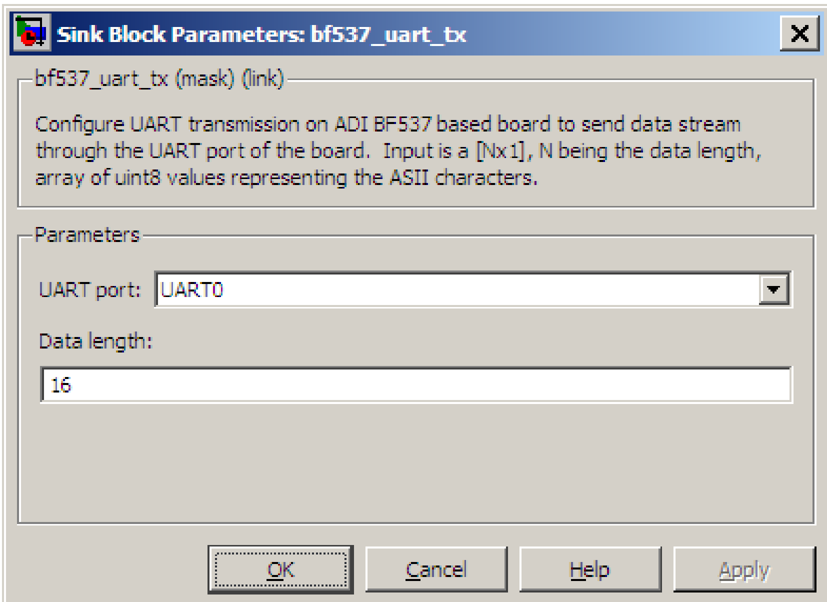
Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ ADSP-BF537 EZ-KIT Lite

Description



Configure UART transmission on ADI BF537 based board to send data stream through the UART port of the board. The block requires an input of [Nx1], where N indicates the data length, in an array of uint8 values representing the ASCII characters. Your model can only contain one transmit block per UART port.

Dialog Box



UART port

Select the UART port the transmit block uses to send data.

Data length

Set the data length, in data words, of each transmission. Match this value to the data size on the **In** port.

References

ADSP-BF537 EZ-KIT Lite® Evaluation System Manual, Part Number 82-000865-01, available from the Analog Devices Web site.

See Also

Blackfin537 bf537_uart_config, Blackfin537 bf537_uart_rx

Blackfin Hardware Interrupt

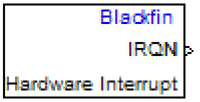
Purpose

Generate Interrupt Service Routine

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ Scheduling

Description



Hardware Interrupt

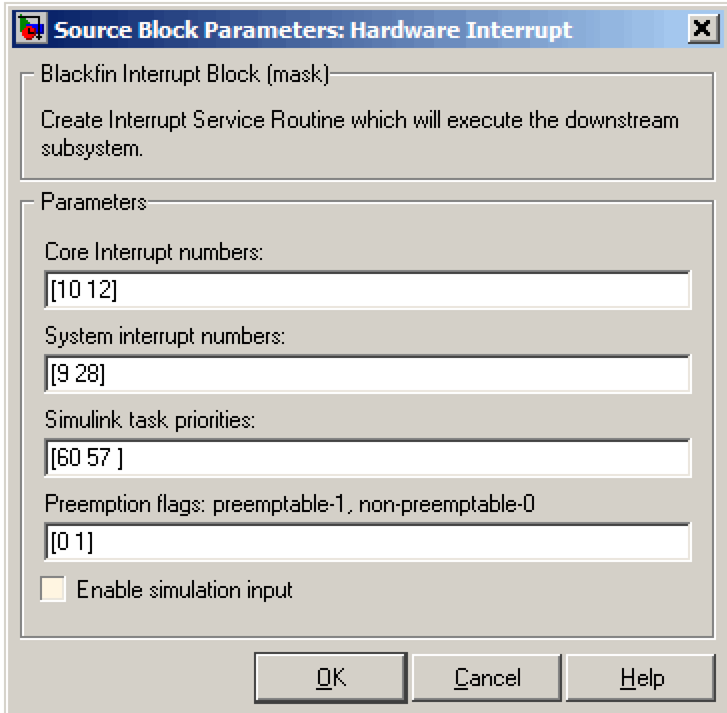
Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from this block or an Idle Task block connected to this block. Core interrupts trigger the ISRs. System interrupts trigger the core interrupts. In the following figure, you see the mapping possibilities between system interrupts and core interrupts.

Interrupts

Blackfin processors support the interrupt numbers shown in the following table. Some Blackfin processors do not support all of the system interrupts.

Interrupt Description	Valid Range in Parameter
Core interrupt numbers	7 to 13 and 15
System interrupt numbers	0 to 63 (The upper end value depends on the processor. May be less than 63.)

Dialog Box



Core interrupt numbers

Specify a vector of one or more interrupt numbers for the interrupt service routines (ISR) to install. The valid range is 7 to 13, and 15, where 7 through 13 are hardware driven, 15 is software driven. Both Green Hills MULTI and Analog Devices VisualDSP++ use core interrupt 14 to service synchronous rates. Core interrupts numbered 0 to 6 are reserved and cannot be entered in this field.

The width of the block output signal corresponds to the number of interrupt values you specify in this field. Triggering of each ISR depends on the core interrupt value, the system interrupt value, and the preemption flag you enter for each interrupt. These three values define how the code and processor respond to interrupts during asynchronous scheduler operations.

System interrupt numbers

System interrupt numbers identify system interrupts to map to core interrupts. Enter one or more values as a vector. The valid range depends on your processor. Some processors do not support the full range of 64 system interrupts. The software does not test for valid system interrupt values. You must verify that your values are valid for your processor. You must specify at least one system interrupt number to use asynchronous scheduling.

The block maps the first interrupt value in this field to the first core interrupt value you enter in **Core interrupt numbers**, it maps the second system interrupt value to the second core interrupt value, and so on until it has mapped all of the system interrupt values to core interrupt values. You cannot map more than one system interrupt to the same core interrupt. Therefore, you can enter one system interrupt value in this field and map it to more than one core interrupt. You cannot enter more than one value in this field and map the values to one core interrupt.

When you trigger one of the system interrupts in this field, the block triggers the ISR associated with the core interrupt that is mapped to the system interrupt.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink task priority specifies the Simulink priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Proper code generation requires rate transition code (see Rate Transitions and Asynchronous Blocks). The task priority values ensure 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: preemptable – 1, non-preemptable – 0

Higher priority interrupts can preempt interrupts that have lower priority. To control this preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering `1` indicates the corresponding core interrupt can be preempted.
- Entering `0` indicates the corresponding interrupt cannot be preempted.

When **Core interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of preemption flag values that correspond to the order of the interrupts in **Core interrupt numbers**. If **Core interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

For example, the default settings `[0 1]`indicate that the interrupt with value `10` in **Core interrupt numbers** is not preemptible and the value `12` interrupt can be preempted.

Enable simulation input

When you select this option, Simulink adds an input port to the Hardware Interrupt block. This port receives input only during simulation. Connect one or more simulated interrupt sources to the simulation input.

Byte Pack

Purpose

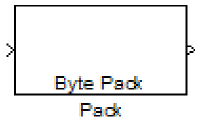
Convert input signals to `uint8` vector

Library

Embedded Coder/ Embedded Targets/ Host Communication

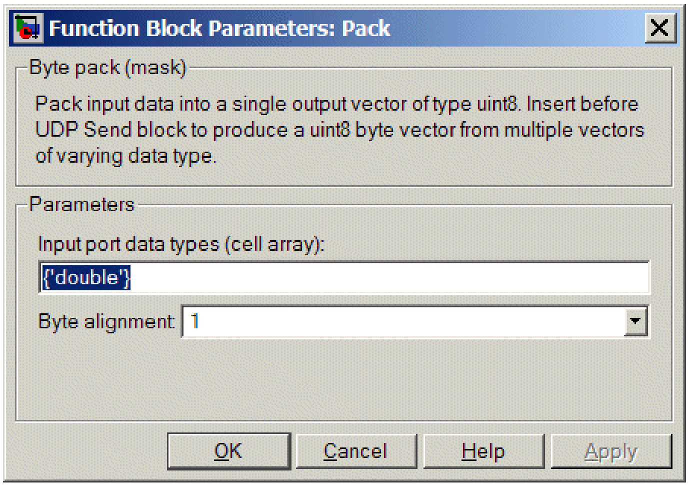
Simulink Coder/ Desktop Targets/ Host Communication

Description



Using the input port, the block converts data of one or more data types into a single `uint8` vector for output. With the options available, you specify the input data types and the alignment of the data in the output vector. Because UDP messages are in `uint8` data format, use this block before a UDP Send block to format the data for transmission using the UDP protocol.

Dialog Box



Input port data types (cell array)

Specify the data types for the different signals as part of the parameters. The block supports all Simulink data types except characters. Enter the data types as Simulink types in the cell array, such as '`double`' or '`int32`'. The order of the data type entries in the cell array must match the order in which the data arrives at the block input. This block determines the signal sizes automatically. The block always has at least one input port and only one output port.

Byte alignment

This option specifies how to align the data types to form the `uint8` output vector. Select one of the values in bytes from the list.

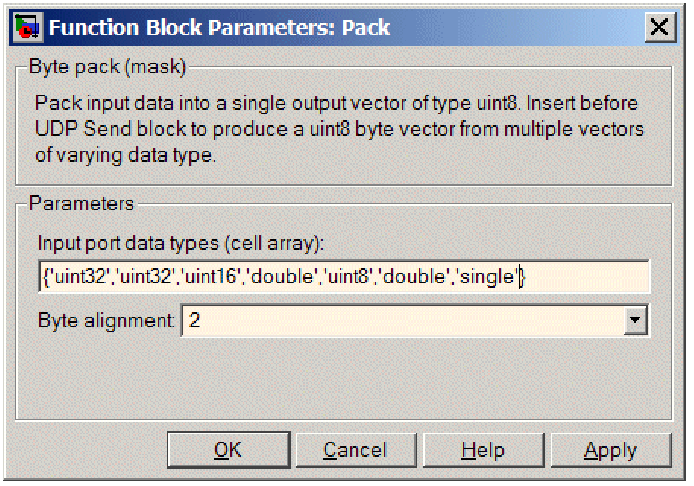
Alignment can occur on 1, 2, 4, or 8-byte boundaries depending on the value you choose. The value defaults to `1`. Given the alignment value, each signal data value begins on multiples of the alignment value. The alignment algorithm ensures that each element in the output vector begins on a byte boundary specified by the alignment value. Byte alignment sets the boundaries relative to the starting point of the vector.

Selecting 1 for **Byte alignment** provides the tightest packing, with no holes between any data types for any combination of data types and signals.

Sometimes, you can have multiple data types of varying lengths. In such cases, specifying a 2-byte alignment can produce 1–byte gaps between uint8 or int8 values and another data type. In the pack implementation, the block copies data to the output data buffer 1 byte at a time. You can specify any of the data alignment options with any of the data types.

Example

Use a cell array to enter input data types in the **Input port data types** parameter. The order of the data types you enter must match the order of the data types at the block input.



In the cell array, you provide the order in which the block expects to receive data—`uint32`, `uint32`, `uint16`, `double`, `uint8`, `double`, and `single`. With this information, the block automatically provides the proper number of input ports.

Byte alignment equal to `2` specifies that each new value begins 2 bytes from the previous data boundary.

The example shows the following data types:

```
{'uint32','uint32','uint16','double','uint8','double','single'}
```

When the signals are scalar values (no matrices or vectors in this example), the first signal value in the vector starts at 0 bytes. Then, the second signal value starts at 2 bytes, and the third at 4 bytes. Next, the fourth signal value follows at 6 bytes, the fifth at 8 bytes, the sixth at 10 bytes, and the seventh at 12 bytes. As the example shows, the packing algorithm leaves a 1-byte gap between the `uint8` data value and the double value.

See Also

Byte Reversal, Byte Unpack

Purpose

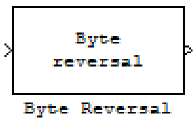
Reverse order of bytes in input word

Library

Embedded Coder/ Embedded Targets/ Host Communication

Simulink Coder/ Desktop Targets/ Host Communication

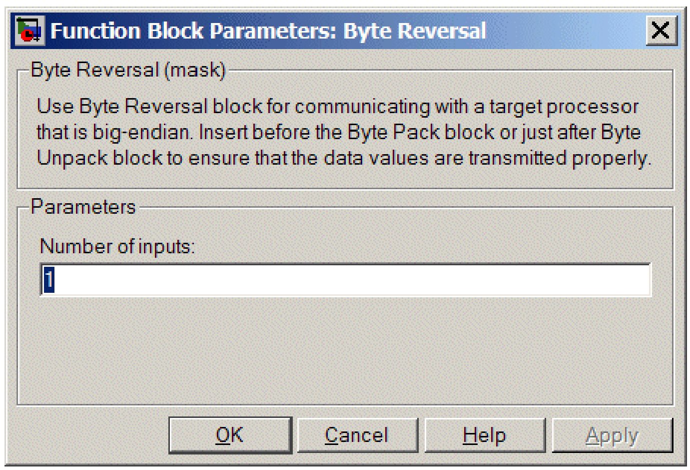
Description



Byte reversal changes the order of the bytes in data you input to the block. Use this block when your process communicates between targets that use different endianness, such as between Intel® processors that are little endian and others that are big endian. Texas Instruments processors are little-endian by default.

To exchange data with a processor that has different endianness, place a Byte Reversal block just before the send block and immediately after the receive block.

Dialog Box



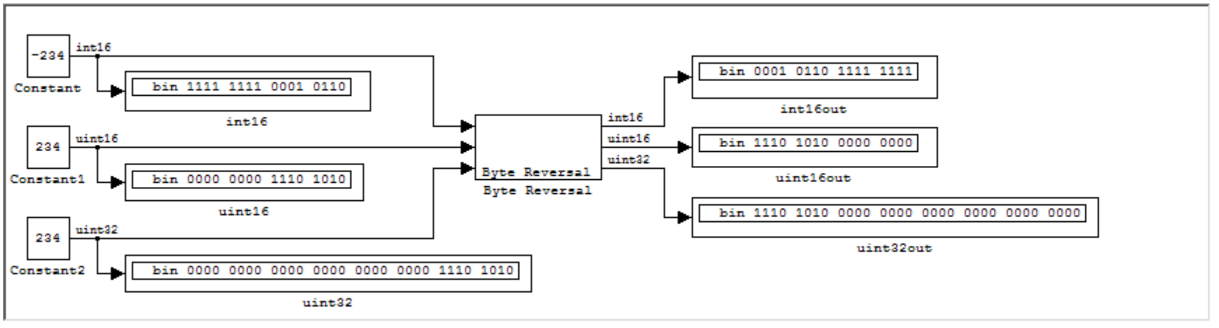
Number of inputs

Specify the number of input ports for the block. The number of input ports adjusts automatically to match value so the number of outputs equals the number of inputs.

When you use more than one input port, each input port maps to the matching output port. Data entering input port 1 leaves through output port 1, and so on.

Reversing the bytes does not change the data type. Input and output retain matching data type.

The following model shows byte reversal in use. In this figure, the input and output ports match for each path.



See Also

Byte Pack, Byte Unpack

Purpose

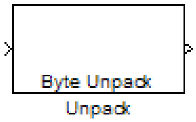
Unpack UDP `uint8` input vector into Simulink data type values

Library

Embedded Coder/ Embedded Targets/ Host Communication

Simulink Coder/ Desktop Targets/ Host Communication

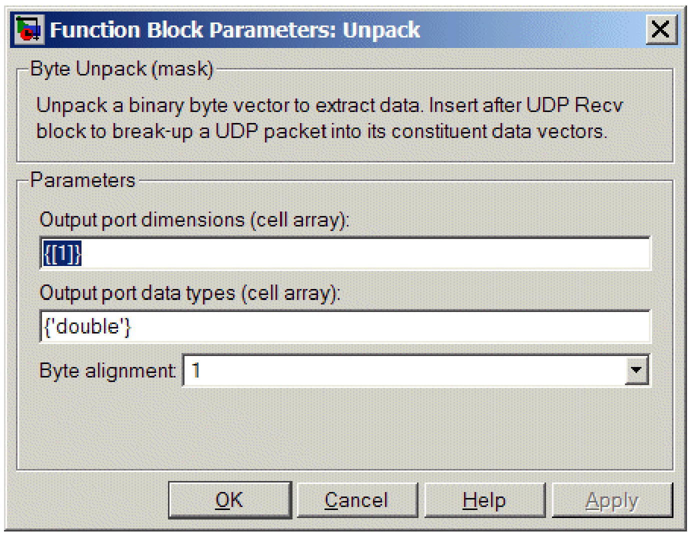
Description



Byte Unpack is the inverse of the Byte Pack block. It takes a UDP message from a UDP receive block as a `uint8` vector, and outputs Simulink data types in various sizes depending on the input vector.

The block supports all Simulink data types.

Dialog Box



Output port dimensions (cell array)

Containing a cell array, each element in the array specifies the dimension that the MATLAB `size` function returns for the corresponding signal. Usually you use the same dimensions as you set for the corresponding Byte Pack block in the model.

Entering one value means that the block applies that dimension to all data types.

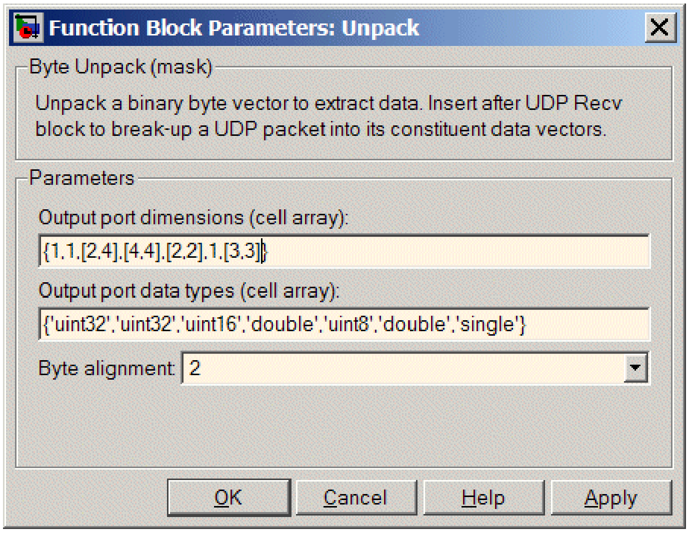
Output port data types (cell array)
Specify the data types for the different input signals to the Pack block. The block supports all Simulink data types—`single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`, and `Boolean`. The entry here is the same as the Input port data types parameter in the Byte Pack block in the model. You can enter one data type and the block applies that type to all output ports.

Byte Alignment
This option specifies how to align the data types to form the input `uint8` vector. Match this setting with the corresponding Byte Pack block alignment value of `1`, `2`, `4`, or `8` bytes.

Example

This figure shows the Byte Unpack block that corresponds to the example in the Byte Pack example. The **Output port data types (cell array)** entry shown is the same as the **Input port data types (cell array)** entry in the Byte Pack block

```
{'uint32','uint32','uint16','double','uint8','double','single'}.
```



In addition, the **Byte alignment** setting matches as well. **Output port dimensions (cell array)** now includes scalar values and matrices to demonstrate entering nonscalar values. The example for the Byte Pack block assumed only scalar inputs.

See Also

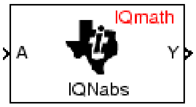
Byte Pack, Byte Reversal

C2000 Absolute IQN

Purpose Absolute value

Library Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

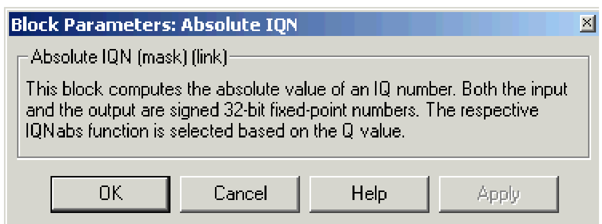
Description This block computes the absolute value of an IQ number input. The output is also an IQ number.



Absolute IQN

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also c2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

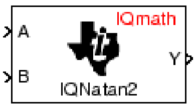
Purpose

Four-quadrant arc tangent

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

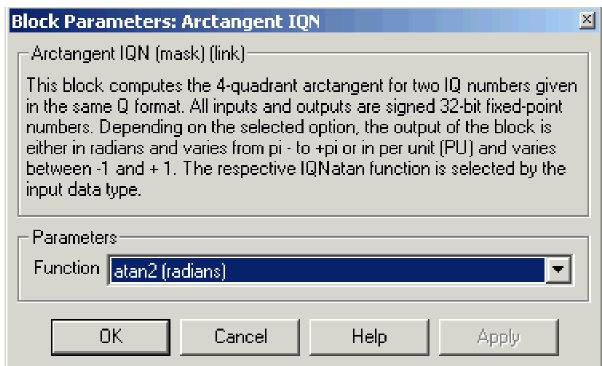


Arctangent IQN

The Arctangent IQN block computes the four-quadrant arc tangent of the IQ number inputs and produces IQ number output.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Function

Type of arc tangent to calculate:

- `atan2` — Compute the four-quadrant arc tangent with output in radians with values from -pi to +pi.
- `atan2PU` — Compute the four-quadrant arc tangent per unit. If `atan2(B,A)` is greater than or equal to 0, `atan2PU(B,A) = atan2(B,A)/2*pi`. Otherwise, `atan2PU(B,A)`

= `atan2(B,A)/2*pi+1`. The output is in per-unit radians with values from 0 to 2*pi radians.

Note The order of the inputs to the Arctangent IQN block correspond to the Texas Instruments convention, with argument 'A' at the top and 'B' at bottom.

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

Purpose

Analog-to-Digital Converter (ADC)

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Description

ADC

The ADC block configures the ADC to perform analog-to-digital conversion of signals connected to the selected ADC input pins. The ADC block outputs digital values representing the analog input signal and stores the converted values in the result register of your digital signal processor. You use this block to capture and digitize analog signals from external sources such as signal generators, frequency generators, or audio devices. With the C28x3x, you can configure the ADC to use the processor's DMA module to move data directly to memory without using the CPU. This frees the CPU to perform other tasks and increases overall system performance.

Output

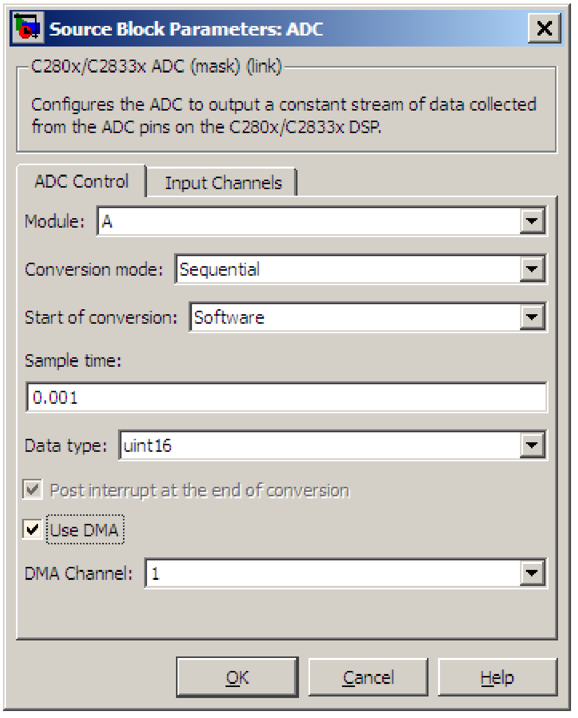
The output of the ADC is a vector of `uint16` values. The output values are in the range 0 to 4095 because the ADC is 12-bit converter.

Modes

The ADC block supports ADC operation in dual and cascaded modes. In dual mode, either module `A` or module `B` can be used for the ADC block, and two ADC blocks are allowed in the model. In cascaded mode, both module `A` and module `B` are used for a single ADC block.

Dialog Box

ADC Control Pane



Module

Specifies which DSP module to use:

- A — Displays the ADC channels in module A (ADCINA0 through ADCINA7).
- B — Displays the ADC channels in module B (ADCINB0 through ADCINB7).
- A and B — Displays the ADC channels in both modules A and B (ADCINA0 through ADCINA7 and ADCINB0 through ADCINB7).

Conversion mode

Type of sampling to use for the signals:

- `Sequential` — Samples the selected channels sequentially.
- `Simultaneous` — Samples the corresponding channels of modules A and B at the same time.

Start of conversion

Type of signal that triggers conversions to begin:

- `Software` — Signal from software. Conversion values are updated at each sample time.
- `ePWMxA` / `ePWMxB` / `ePWMxA_ePWMxB` — Start of conversion is controlled by user-defined PWM events.
- `XINT2_ADCSOC` — Start of conversion is controlled by the `XINT2_ADCSOC` external signal pin.

The choices available in **Start of conversion** depend on the **Module** setting. The following table summarizes the available choices. For each set of **Start of conversion** choices, the default is given first.

Module Setting	Start of Conversion Choices
A	Software, ePWMxA, XINT2_ADCSOC
B	ePWMxB, Software
A and B	Software, ePWMxA, ePWMxB, ePWMxA_ePWMxB, XINT2_ADCSOC

Sample time

Time in seconds between consecutive sets of samples that are converted for the selected ADC channel(s). This is the rate at which values are read from the result registers. To execute this block asynchronously, set **Sample Time** to `-1`, check the **Post interrupt at the end of conversion** box, and refer to

"Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

To set different sample times for different groups of ADC channels, you must add separate ADC blocks to your model and set the desired sample times for each block.

Data type
Date type of the output data. Valid data types are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

Post interrupt at the end of conversion
Select this check box to post an asynchronous interrupt at the end of each conversion. The interrupt is always posted at the end of conversion. To execute this block asynchronously, set **Sample Time** to `-1`, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

Use DMA (with C28x3x)
Enable the Direct Memory Access (DMA) to transfer data directly from the ADC to memory, bypassing the CPU and improving overall system performance. This feature is only valid with a C28x3x target.

When enabled, this setting applies the following settings to the channel specified by the **DMA Channel** parameter. *Disable* the corresponding channel in the **Target Preferences block > Peripherals > DMA_ch#**. Modifications to **Target Preferences block > Peripherals > DMA_ch#** do not apply or override the following settings:

- **Enable DMA channel**: Enabled for channel specified by the ADC block **DMA Channel** parameter.
- **Data size**: `16 bit`
- **Interrupt source**: If the ADC block **Module** is `A` or `A and B`, **Interrupt source** is `SEQ1INT`. If the ADC block **Module** is `B`, **Interrupt source** is `SEQ2INT`.

- **Generate interrupt**: `Generate interrupt at end of transfer`
- **Size**
  - **Burst**: The value assigned to **Burst** equals the ADC block **Number of conversions** (NOC) multiplied by a value for the ADC block **Conversion mode** (CVM). To summarize, **Burst** = NOC * CVM.

    If **Conversion mode** is `Sequential`, CVM = 1. If **Conversion mode** is `Simultaneous`, CVM = 2.

    For example, **Burst** is 6 when NOC is 3 and CVM is 2.
  - **Transfer**: 1
  - **SRC wrap**: 65536
  - **DST wrap**: 65536
- **Source**
  - **Begin address**: The value of **Begin address** is 0xB00 if the ADC block **Module** is `A` or `A and B`. The value of **Begin address** is 0xB08 if the ADC block **Module** is `B`.
  - **Burst step**: 1
  - **Transfer step**: 0
  - **Wrap step**: 0
- **Destination**
  - **Begin address**: The value of **Begin address** is the ADC buffer address minus the ADC block **Number of conversions**.

    If the target is F28232 or F28332, the ADC buffer address is 0xDFFC (57340). For other C28x3x targets, the ADC buffer address is 0xFFFC (65532).

For example, with a F28232 target, the **Begin address** is 0xDFF9 (57337) because the ADC buffer address, 57340 (0xDFFC), minus 3 conversions equals 57337 (0xDFF9).

- **Burst step**: 1
- **Transfer step**: 1
- **Wrap step**: 0

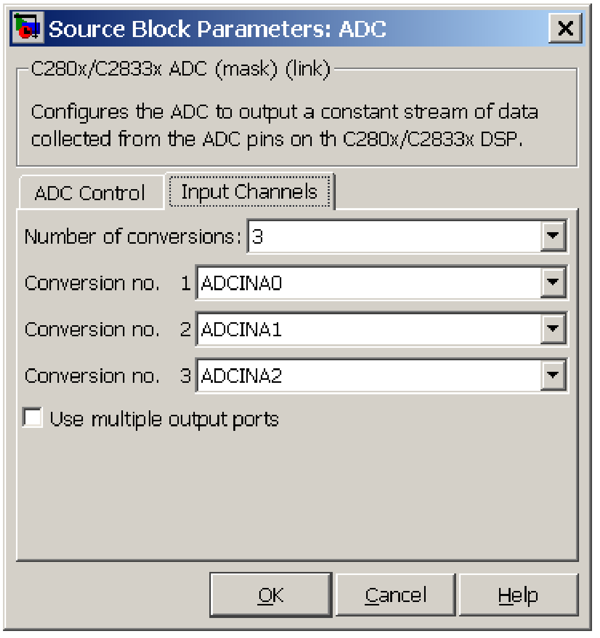
- **Mode**
  - **Enable one shot mode**: disabled
  - **Sync enable**: disabled
  - **Enable continuous mode**: enabled
  - **Enable DST sync mode**: disabled
  - **Set channel 1 to highest priority**: disabled
  - **Enable overflow interrupt**: disabled

For more information, consult *TMS320x2833x, 2823x Direct Memory Access (DMA) Module Reference Guide, Literature Number: SPRUFB8A*, available at the Texas Instruments Web site.

DMA Channel

When the **Use DMA** parameter is enabled, select a channel for the DMA module to use for data transfers. To prevent channel conflicts, the same channel number must remain disabled in the Target Preferences block, otherwise the software will generate an error message.

Input Channels Pane



Number of conversions

Number of ADC channels to use for analog-to-digital conversions.

Conversion no.

Specific ADC channel to associate with each conversion number.

In oversampling mode, a signal at a given ADC channel can be sampled multiple times during a single conversion sequence. To oversample, specify the same channel for more than one conversion. Converted samples are output as a single vector.

Use multiple output ports

If more than one ADC channel is used for conversion, you can use separate ports for each output and show the output ports on the block. If you use more than one channel and do not use multiple output ports, the data is output in a single vector.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x ePWM

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

“Configuring Acquisition Window Width for ADC Blocks”

“ADC” on page 5-883

Purpose

Implement CAN Calibration Protocol (CCP) standard

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description

CAN Calibration Protocol

The CAN Calibration Protocol block provides an implementation of a subset of the CAN Calibration Protocol (CCP) Version 2.1. CCP is a protocol for communicating between the target processor and the host machine over CAN. In particular, a calibration tool (see "Compatibility with Calibration Packages" on page 5-42) running on the host can communicate with the target, allowing remote signal monitoring and parameter tuning.

This block processes a Command Receive Object (CRO) and outputs the resulting Data Transmission Object (DTO) and Data Acquisition (DAQ) messages.

For more information on CCP, refer to *ASAM Standards: ASAM MCD: MCD 1a* on the Association for Standardization of Automation and Measuring Systems (ASAM) Web site at `http://www.asam.de`.

Using the DAQ Output

Note The CCP Data Acquisition (DAQ) List mode of operation is only supported with Embedded Coder. If Embedded Coder is not available then custom storage classes `canlib.signal` are ignored during code generation: this means that the CCP DAQ Lists mode of operation cannot be used.

You can use the CCP Polling mode of operation with or without Embedded Coder.

The DAQ output is the output for any CCP Data Acquisition (DAQ) lists that have been set up. You can use the ASAP2 file generation feature of the Real-Time (RT) target to

- Set up signals to be transmitted using CCP DAQ lists.
- Assign signals in your model to a CCP event channel automatically (see "Generating an ASAP2 File").

Once these signals are set up, event channels then periodically fire events that trigger the transmission of DAQ data to the host. When this occurs, CAN messages with the appropriate CCP/DAQ data appear on the DAQ output, along with an associated function call trigger.

The calibration tool (see "Compatibility with Calibration Packages" on page 5-42) must use CCP commands to assign an event channel and data to the available DAQ lists, and interpret the synchronous response.

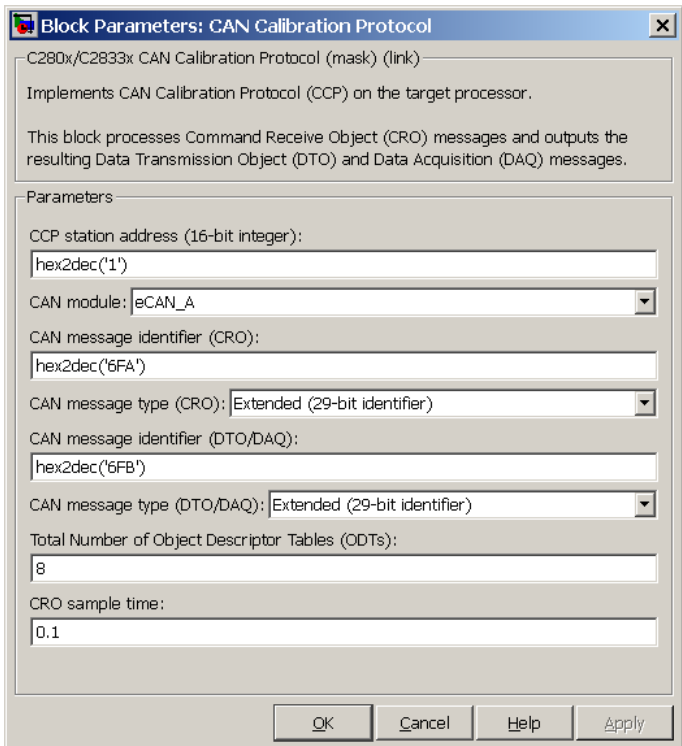
Using DAQ lists for signal monitoring has the following advantages over the polling method:

- There is no need for the host to poll for the data. Network traffic is halved.
- The data is transmitted at the correct update rate for the signal. Therefore, there is no unnecessary network traffic generated.

- Data is consistent. The transmission takes place after the signals have been updated, so there is no risk of interruptions while sampling the signal.

Note Embedded Coder software does not currently support event channel prescalers.

Dialog Box



CCP station address (16–bit integer)

The station address of the target. The station address is interpreted as a `uint16`. It is used to distinguish between different targets. By assigning unique station addresses to targets sharing the same CAN bus, it is possible for a single host to communicate with multiple targets.

CAN module
If your processor has more than one module, select the module this block configures.

CAN message identifier (CRO)
Specify the CAN message identifier for the Command Receive Object (CRO) message you want to process.

CAN message type (CRO)
The incoming message type. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

CAN message identifier (DTO/DAQ)
The message identifier is the CAN message ID used for Data Transmission Object (DTO) and Data Acquisition (DAQ) message outputs.

CAN message type (DTO/DAQ)
The message type to be transmitted by the DTO and DAQ outputs. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

Total Number of Object Descriptor Tables (ODTs)
The default number of Object Descriptor Tables (ODTs) is 8. These ODTs are shared equally between all available DAQ lists. You can choose a value between 0 and 254, depending on how many signals you log simultaneously. You must make sure you allocate at least 1 ODT per DAQ list, or your build will fail. The calibration tool will give an error message if there are too few ODTs for the number of signals you specify for monitoring. Be aware that too many ODTs can make the sample time overrun. If you choose more than the maximum number of ODTs (254), the build will fail.

A single ODT uses 56 bytes of memory. Using all 254 ODTs would require over 14 KB of memory, a large proportion of the available memory on the target. To conserve memory on the target, the default number is low, allowing DAQ list signal monitoring with reduced memory overhead and processing power.

As an example, if you have five different rates in a model, and you are using three rates for DAQ, then this will create three DAQ lists and you must make sure you have at least three ODTs. ODTs are shared equally among DAQ lists and, therefore, you will end up with one ODT per DAQ list. With less than three ODTs, you get zero ODTs per DAQ list and the behavior is undefined.

Taking this example further, say you have three DAQ lists with one ODT each, and start trying to monitor signals in a calibration tool. If you try to assign too many signals to a particular DAQ list (that is, signals requiring more space than seven bytes (one ODT) in this case), then the calibration tool will report this as an error.

CRO sample time
The sample time for CRO messages.

Supported CCP Commands

The following CCP commands are supported by the CAN Calibration Protocol block:

- `CONNECT`
- `DISCONNECT`
- `DNLOAD`
- `DNLOAD_6`
- `EXCHANGE_ID`
- `GET_CCP_VERSION`
- `GET_DAQ_SIZE`
- `GET_S_STATUS`
- `SET_DAQ_PTR`

- SET_MTA
- SET_S_STATUS
- SHORT_UP
- START_STOP
- START_STOP_ALL
- TEST
- UPLOAD
- WRITE_DAQ

Compatibility with Calibration Packages

The above commands support

- Synchronous signal monitoring via calibration packages that use DAQ lists
- Asynchronous signal monitoring via calibration packages that poll the target
- Asynchronous parameter tuning via CCP memory programming

This CCP implementation has been tested successfully with the Vector-Informatik CANape calibration package running in both DAQ list and polling mode, and with the Accurate Technologies, Inc., Vision, calibration package running in DAQ list mode. (Accurate Technologies, Inc., Vision does not support the polling mechanism for signal monitoring).

Purpose

Enhanced Control Area Network receive mailbox

Library

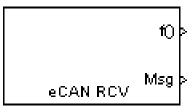
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



eCAN Receive

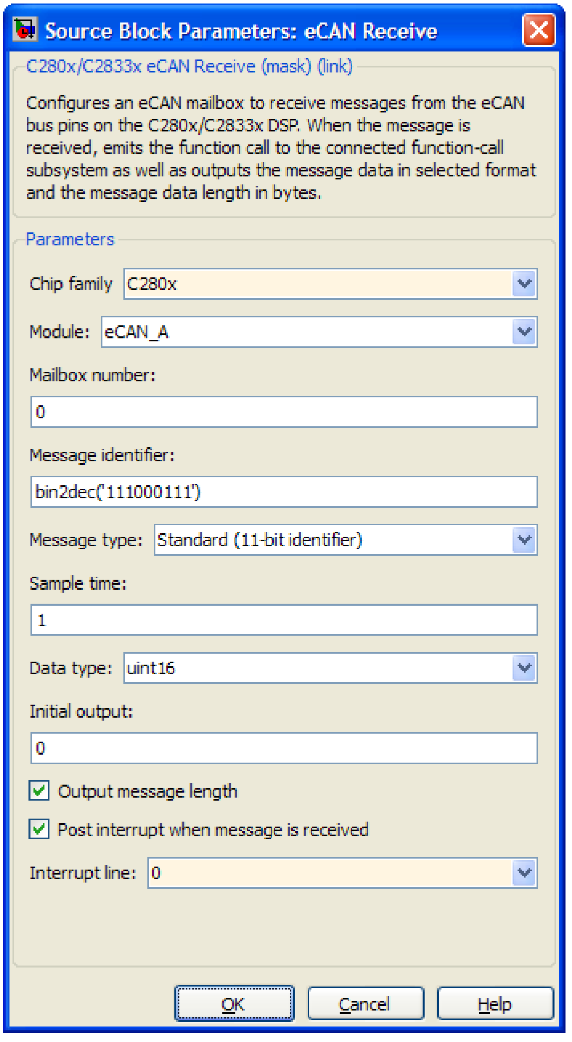
The C280x/C2803x/C28x3x enhanced Control Area Network (eCAN) Receive block generates source code for receiving eCAN messages through an eCAN mailbox. The eCAN modules on the DSP chip provide serial communication capability and have 32 mailboxes configurable for receive or transmit. The C280x/C2803x/C28x3x supports eCAN data frames in standard or extended format.

The eCAN Receive block has up to two and, optionally, three output ports.

- The first output port is the function call port, and a function call subsystem should be connected to this port. When a new message is received, this subsystem is executed.
- The second output port is the message data port. The received data is output in the form of a vector of elements of the selected data type. The length of the vector is always 8 bytes. The message data port will always output data. When the block is used in polling mode, if there is no new message created between the consecutive executions of the block, then the old message, or the existing message, is repeated.
- The third output port is optional and appears only if **Output message length** is selected.

To use the eCAN Receive block with the eCAN Pack block in the `canmsglib`, set **Data type** to `CAN_MESSAGE_TYPE`.

Dialog Box



Chip family
Select the processor that has the eCAN module.

Module

Determines which of the two eCAN modules is being configured by this instance of the eCAN Receive block. Options are `eCAN_A` and `eCAN_B`.

This parameter is not visible when you set **Chip family** to `C2803x`.

Mailbox number

Sets the value of the mailbox number register (MBNR). For standard CAN controller (SCC) mode, enter a unique number from `0` to `15`. For high-end CAN controller (HECC) mode enter a unique number from `0` to `31` . In SCC mode, transmissions from the mailbox with the highest number have the highest priority. In HECC mode, the mailbox number only determines priority if the Transmit priority level (TPL) of two mailboxes is equal.

Message identifier

Sets the value of the message identifier register (MID). The message identifier is 11 bits long for standard frame size or 29 bits long for extended frame size in decimal, binary, or hex format. For the binary and hex formats, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry.

Message type

Select `Standard (11-bit identifier)` or `Extended (29-bit identifier)`.

Sample time

Frequency with which the mailbox is polled to determine if a new message has been received. A new message causes a function call to be emitted from the mailbox. If you want to update the message output only when a new message arrives, then the block needs to be executed asynchronously. To execute this block asynchronously, set **Sample Time** to `-1`, check the **Post interrupt when message is received** box, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

Note For information about setting the timing parameters of the CAN module, see "Configuring Timing Parameters for CAN Blocks".

Data type

Select one of the following options:

- `uint8` (vector length = 8 elements)
- `uint16` (vector length = 4 elements)
- `uint32` (vector length = 2 elements)
- `CAN_MESSAGE_TYPE` (Select this option to use the eCAN receive block with the CAN Unpack block.)

The length of the vector for the received message is at most 8 bytes. If the message is less than 8 bytes, the data buffer bytes are right-aligned in the output. The data are unpacked as follows using the data buffer, which is 8 bytes.

For `uint8` data, eCAN Receive reads each unit of 8 bytes in the registers, and outputs 8-bit data to 8 elements (using the lower part of the 16-bit memory):

```
Output[0] = data_buffer[0];
Output[1] = data_buffer[1];
Output[2] = data_buffer[2];
Output[3] = data_buffer[3];
Output[4] = data_buffer[4];
Output[5] = data_buffer[5];
Output[6] = data_buffer[6];
Output[7] = data_buffer[7];
```

For `uint16` data,

```
Output[0] = data_buffer[1..0];
Output[1] = data_buffer[3..2];
```

```
Output[2] = data_buffer[5..4];
Output[3] = data_buffer[7..6];
```

For `uint32` data,

```
Output[0] = data_buffer[3..0];
Output[1] = data_buffer[7..4];
```

For example, if the received message has two bytes:

```
data_buffer[0] = 0x21
data_buffer[1] = 0x43
```

The `uint16` output would be:

```
Output[0] = 0x4321
Output[1] = 0x0000
Output[2] = 0x0000
Output[3] = 0x0000
```

When you select CAN_MESSAGE_TYPE, the block outputs the following struct data (defined in can_message.h):

```
struct {

  /* Is Extended frame */
  uint8_T Extended;

  /* Length */
  uint8_T Length;

  /* RTR */
  uint8_T Remote;

  /* Error */
  uint8_T Error;
```

```
    /* CAN ID */
    uint32_T ID;

    /*
    TIMESTAMP_NOT_REQUIRED is a macro that will be defined by Target teams
    PIL, xPC if they do not require the timestamp field during code
    generation. By default, timestamp is defined. If the targets do not require
    the timestamp field, they should define the macro TIMESTAMP_NOT_REQUIRED before
    including this header file for code generation.
    */
    #ifndef TIMESTAMP_NOT_REQUIRED
      /* Timestamp */
      double Timestamp;
    #endif

    /* Data field */
    uint8_T Data[8];

  };
```

Initial output
Set the value the eCAN node outputs to the model before it has received any data. The default value is `0`.

Output message length
Select to output the message length in bytes to the third output port. If not selected, the block has only two output ports.

Post interrupt when message is received
Select this check box to post an asynchronous interrupt when a message is received.

Interrupt line
Select the interrupt line the asynchronous interrupt uses. This action sets bit 2 (GIL) in the Global Interrupt Mask Register (CANGIM):

- `1` maps the global interrupts to the ECAN1INT line.

- 0 maps the global interrupts to the ECAN0INT line.

References

For detailed information on the eCAN module, visit ti.com and search for the documentation related to your processor. The following materials are available at the Texas Instruments Web site:

- *TMS320F2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide*, Literature Number SPRUEU1
- *TMS320x280x/2801x Enhanced Controller Area Network (eCAN) Reference Guide*, Literature Number SPRUEU0
- *TMS320x2803x Piccolo Enhanced Controller Area Network (eCAN) Reference Guide*, Literature Number: SPRUGL7

See Also

C280x/C2803x/C28x3x/c2834x eCAN Transmit

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

"eCAN_A, eCAN_B" on page 5-886

C280x/C2803x/C28x3x/c2834x eCAN Transmit

Purpose

Enhanced Control Area Network transmit mailbox

Library

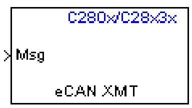
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



eCAN Transmit

The C280x/C2803x/C28x3x enhanced Control Area Network (eCAN) Transmit block generates source code for transmitting eCAN messages through an eCAN mailbox. The eCAN modules on the DSP chip provide serial communication capability and have 32 mailboxes configurable for receive or transmit. The C280x/C2803x/C28x3x supports eCAN data frames in standard or extended format.

Note Fixed-point inputs are not supported for this block.

Data Vectors

The length of the vector for each transmitted mailbox message is 8 bytes. Input data are always right-aligned in the message data buffer. Only `uint16` (vector length = 4 elements) or `uint32` (vector length = 2 elements) data are accepted. The following examples show how the different types of input data are aligned in the data buffer:

For input of type `uint32`,

```
inputdata [0] = 0x12345678
```

the data buffer is:

```
data buffer[0] = 0x78
```

```
data buffer[1] = 0x56
data buffer[2] = 0x34
data buffer[3] = 0x12
data buffer[4] = 0x00
data buffer[5] = 0x00
data buffer[6] = 0x00
data buffer[7] = 0x00
```

For input of type `uint16`,

```
inputdata [0] = 0x1234
```

the data buffer is:

```
data buffer[0] = 0x34
data buffer[1] = 0x12
data buffer[2] = 0x00
data buffer[3] = 0x00
data buffer[4] = 0x00
data buffer[5] = 0x00
data buffer[6] = 0x00
data buffer[7] = 0x00
```

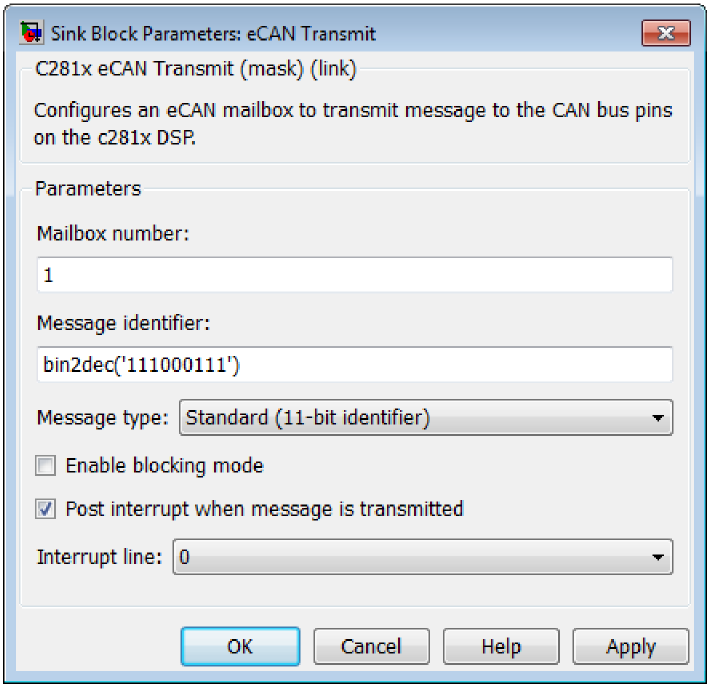
For input of type `uint16[2]`, which is a two-element vector,

```
inputdata [0] = 0x1234
inputdata [1] = 0x5678
```

the data buffer is:

```
data buffer[0] = 0x34
data buffer[1] = 0x12
data buffer[2] = 0x78
data buffer[3] = 0x56
data buffer[4] = 0x00
data buffer[5] = 0x00
data buffer[6] = 0x00
data buffer[7] = 0x00
```

Dialog Box



Module

Determines which of the two eCAN modules is being configured by this instance of the eCAN Transmit block. Options are `eCAN_A` and `eCAN_B`.

Mailbox number

Unique number from `0` to `15` for standard or from `0` to `31` for enhanced CAN mode. It refers to a mailbox area in RAM. In standard mode, the mailbox number determines priority.

Message identifier

Identifier of length 11 bits for standard frame size or length 29 bits for extended frame size in decimal, binary, or hex. If in binary or hex, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry. The message identifier is coded into a message that is sent to the CAN bus.

Message type
Select `Standard (11-bit identifier)` or `Extended (29-bit identifier)`.

Enable blocking mode
If selected, the CAN block code waits indefinitely for a transmit (XMT) acknowledge. If not selected, the CAN block code does not wait for a transmit (XMT) acknowledge, which is useful when the hardware might fail to acknowledge transmissions.

Post interrupt when message is transmitted
If selected, an asynchronous interrupt will be posted when data is transmitted.

Interrupt Line
Select the interrupt line the asynchronous interrupt uses. This action sets bit 2 (GIL) in the Global Interrupt Mask Register (CANGIM):

- `1` maps the global interrupts to the ECAN1INT line.
- `0` maps the global interrupts to the ECAN0INT line.

Note For information about setting the timing parameters of the CAN module, see "Configuring Timing Parameters for CAN Blocks".

References

For detailed information on the eCAN module, see the following materials, available at the Texas Instruments Web site:

- *TMS320F2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide*, Literature Number SPRUEU1
- *TMS320x2803x Piccolo Enhanced Controller Area Network (eCAN) Reference Guide*, Literature Number: SPRUGL7

See Also

C280x/C2803x/C28x3x/c2834x eCAN Receive

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

"eCAN_A, eCAN_B" on page 5-886

C280x/C2802x/C2803x/C28x3x/c2834x eCAP

Purpose

Receive and log capture input pin transitions or configure auxiliary pulse width modulator

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

eCAP

Description

Dialog Box

The eCAP block dialog box provides configuration parameters on four tabbed panes:

- **General**—Set the operating mode for the block (whether the block performs eCAP or APWM processes, assign the pin associated, and set the sample time
- **eCAP**—Configure eCAP functions such as prescaler value, capture pin, and mode control
- **APWM**—Configure waveform and duty cycle values for the pulse width modulation capability
- **Interrupt**—Specify when the block posts interrupts

You can add up to six eCAP blocks to your model, one block for each capture pin. For example, you can have one block configured for eCAP mode with `eCAP1` pin selected and five blocks configured for APWM mode with assigned pins `eCAP2` through `eCAP6`. Or six blocks configured for eCAP mode with each block assigned a different eCAP pin. You cannot assign the same eCAP pin to two eCAP blocks in one model.

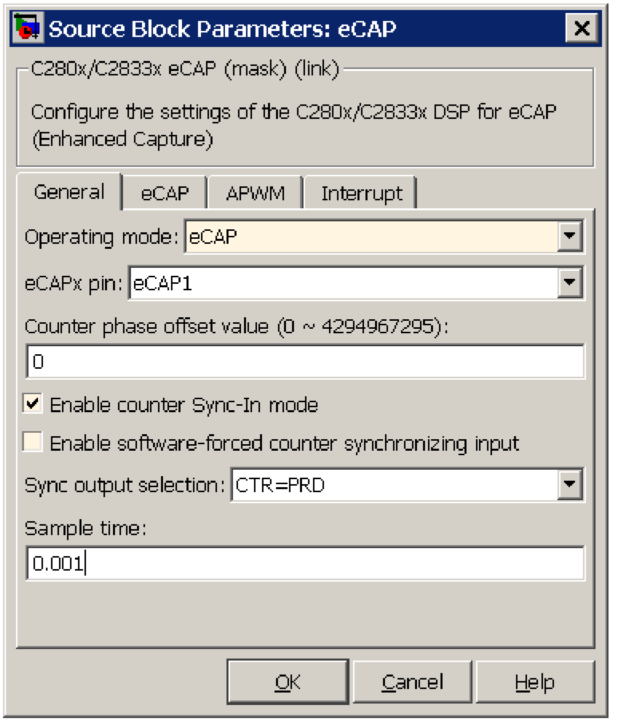
Block Input and Output Ports

The eCAP block has optional input and output ports as shown in the following table.

Port	Description and When the Port is Enabled
Input port SI	Synchronization input for input value from software. Enabled when you select **Enable software forced counter synchronizing input** in either operating mode.
Input port RA	One-shot arming starts the one-shot sequence. Enabled when you set the mode control to `One shot`.
Output port TS	When you enable the reset counter, this option resets the capture event counter after capturing the event time stamp. Enabled when you select **Enable reset counter after capture event 1 time-stamp**.
Output port CF	This port reports the status of the capture event. Enabled when you select **Enable capture event status flag output**.
Output port OF	Enabled when you select **Enable overflow status flag output**.

Note The outputs of this block can be vectorized.

General Pane



Operating mode

When you select `eCAP`, the block captures and logs pin transitions for each capture unit to a FIFO buffer. When you select `APWM`, the block generates asymmetric pulse width modulation (APWM) waveforms for driving downstream systems.

eCAPx pin

The capture unit includes the following features:

- One pin for each capture unit. For example, eCAP1, eCAP2, and so on.
- Four maskable interrupt flags, one for each capture unit.
- Ability to specify the transition detection—rising edge, falling edge, or both edges.

Counter phase offset value (0~4294967295)
: The value you enter here provides the time base for event captures, clocked by the system clock. A phase register is used to synchronize with other counters via the software or hardware forced sync (refer to **Enable counter Sync-In mode**). This is particularly useful in APWM mode when you need a phase offset between capture modules. Enter the phase offset as an integer from 0 (no offset) to 42949667295 (2^{32}) counts.

Enable counter Sync-In mode
: Select this to enable the TSCTR counter to load from the TSCTR register when the block receives either the SYNC1 signal or a software force event (refer to **Enable software-forced counter synchronizing input**).

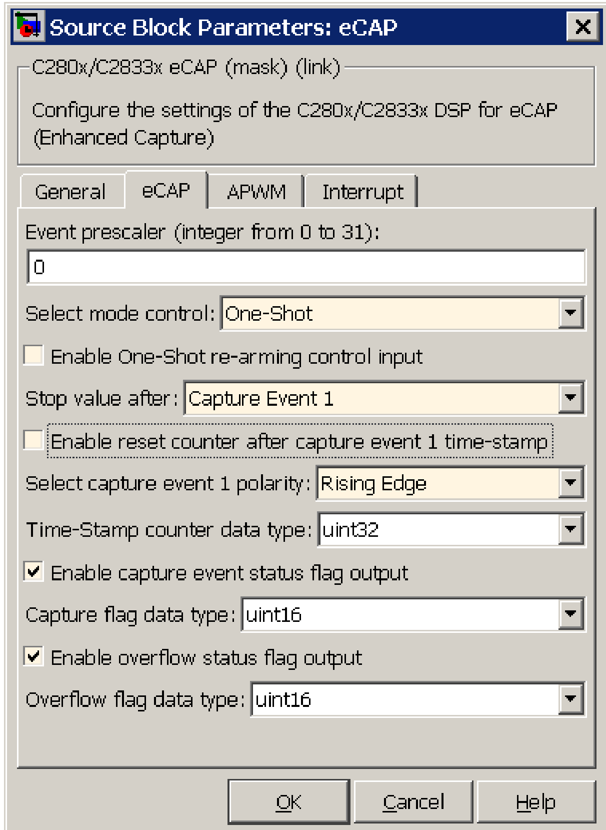
Enable software-forced counter synchronizing input
: This option provides a convenient software method for synchronizing one or more eCAP time bases.

Sync output selection
: Select one of the list entries `Pass through`, `CTR=PRD`, or `Disabled` to synchronize with other counters.

Sample time
: Set the sample time for the block in seconds.

eCAP Pane

To enable the configuration parameters on this pane, select `eCAP` from the **Operating mode** list on the **General** pane.



Event prescaler (integer from 0 to 31)

Multiply the input signal, called a pulse train, by this value. Entering a 0 bypasses the input prescaler, leaving the input capture signal unchanged.

Select mode control

Continuous performs continuous timestamp captures using a circular buffer to capture events 1 through 4.

One-Shot disables continuous mode and enables the **Enable one-shot rearming control via input port** option so you can select it.

Enable one-shot rearming control via input port

Select this option to arm the one-shot sequence:

1. Reset the Mod4 counter to zero.
2. Unfreeze the Mod4 counter.
3. Enable capture register loading.

Stop value after

Specifies the number of capture events after which to stop the capture.

Enable reset counter after capture event 1 timestamp

Enables a reset after capture event 1 and creates an **Output port TS**. When you select this option, the eCAP process resets the counters after receiving a capture event 1 timestamp.

Select capture event 1 polarity

Start the capture event on a **Rising edge** or **Falling edge**.

Time-Stamp counter data type

Select the data type of the counter. The list includes integer and unsigned 8-, 16-, and 32-bit data types, double, single, and Boolean.

Enable capture event status flag output

Output the capture event status flag on the **Output port CF**. The block outputs a 0 until the event capture. After the event, the flag value is 1.

Overflow capture event flag data type

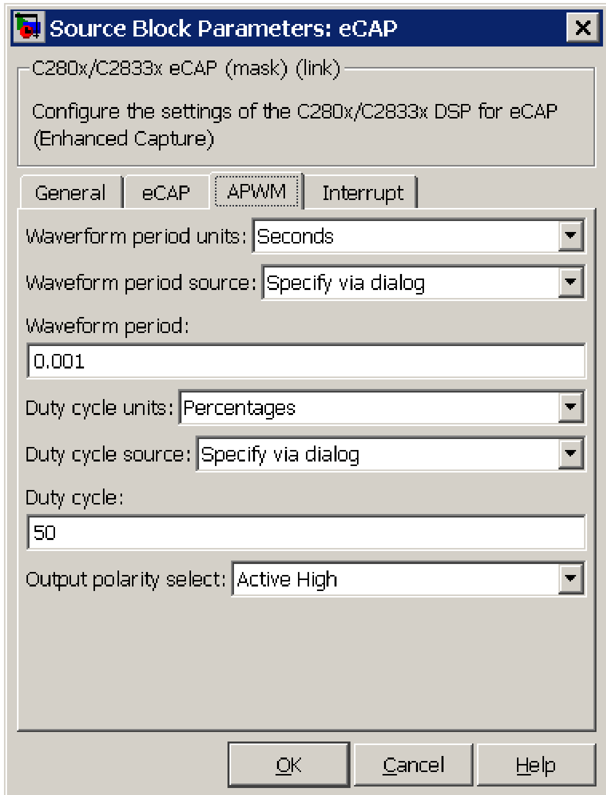
Select the data type to represent the capture event flag. The list includes integer and unsigned 8-, 16-, and 32-bit data types, double, single, and Boolean.

Enable overflow status flag output
Output the status of the elements of the FIFO buffer on the **Output port OF**. After you select this option, set the data type for the flag in **Overflow flag data type**.

Overflow flag data type
Select the data type to represent the status flag. The list includes integer and unsigned 8-, 16-, and 32-bit data types, double, single, and Boolean.

APWM Pane

To enable the configuration parameters on this pane, select `APWM` from the **Operating mode** list on the **General** pane.



Waveform period units

Set the units for measuring the waveform period. **Clock cycles** uses the high-speed peripheral clock cycles of the DSP chip, or `Seconds`. Changing these units changes the **Waveform period** value and the **Duty cycle** value and **Duty cycle units** selection.

Waveform period source

Source from which the waveform period value is obtained. Select `Specify via dialog` to enter the value in **Waveform period** or select `Input port` to use a value from the input port.

Waveform period
Period of the PWM waveform measured in clock cycles or in seconds, as specified in the **Waveform period units**.

Note The term *clock cycles* refers to the high-speed peripheral clock on the F2812 chip. This clock is 75 MHz by default because the high-speed peripheral clock prescaler is set to 2 (150 MHz/2).

Duty cycle units
Units for the duty cycle. Select `Clock cycles` or `Percentages` from the list. Changing these units changes the **Duty cycle** value, the **Waveform period** value, and **Waveform period units** selection.

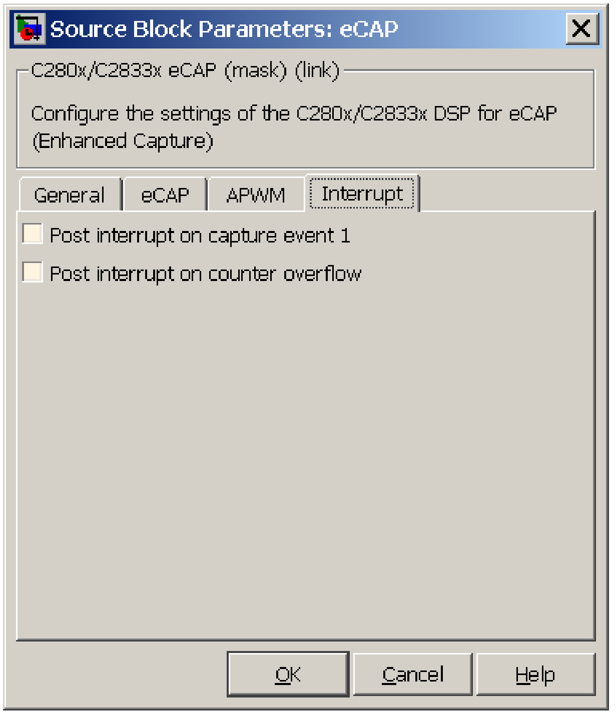
Duty cycle source
Source from which the duty cycle for the specific PWM pair is obtained. Select `Specify via dialog` to enter the value in **Duty cycle** or select `Input port` to use a value from the input port.

Duty cycle
Ratio of the PWM waveform pulse duration to the PWM waveform period expressed in **Duty cycle units**.

Output polarity select
Set the active level for the output. Choose `Active High` or `Active Low` from the list. When you select `Active High`, the compare value defines the high time. Selecting `Active Low` directs the compare value to define the low time.

Interrupt Pane

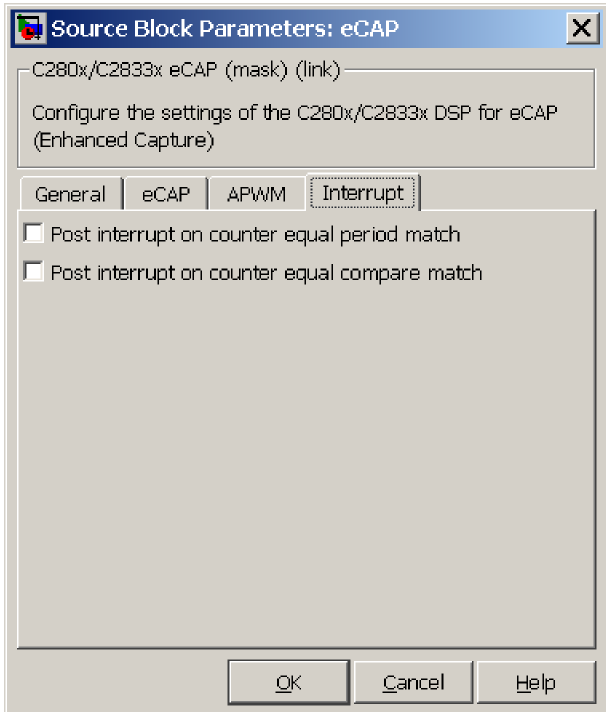
In the following figure, you see the interrupt options when you put the block in eCAP mode by setting **Operating mode** on the **General** pane to `eCAP`.



Post interrupt on capture event 1
Enables capture event 1 as an interrupt source. You can use the C280x/C2802x/C2803x/C28x3x Hardware Interrupt block to react to this interrupt.

Post interrupt on counter overflow
Enables counter overflow as an interrupt source.

The next figure presents the interrupt options when you put the block in APWM mode by setting **Operating mode** on the **General** pane to `APWM`.



Post interrupt on counter equal period match
: Post an interrupt when the value of the counter is the same as the value of the period register (CTR=PRD).

Post interrupt on counter equal compare match
: Post an interrupt when the value of the counter is the same as the value of the compare register (CTR=CMP).

References

For detailed information about interrupt processing, see *TMS320x28xx, 28xxx Enhanced Capture (eCAP) Module Reference Guide*, SPRU807B, available at the Texas Instruments Web site.

See Also

"eCAP" on page 5-889

C280x/C2802x/C2803x/C28x3x/c2834x ePWM

Purpose

Configure Event Manager to generate Enhanced Pulse Width Modulator (ePWM) waveforms

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

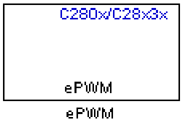
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



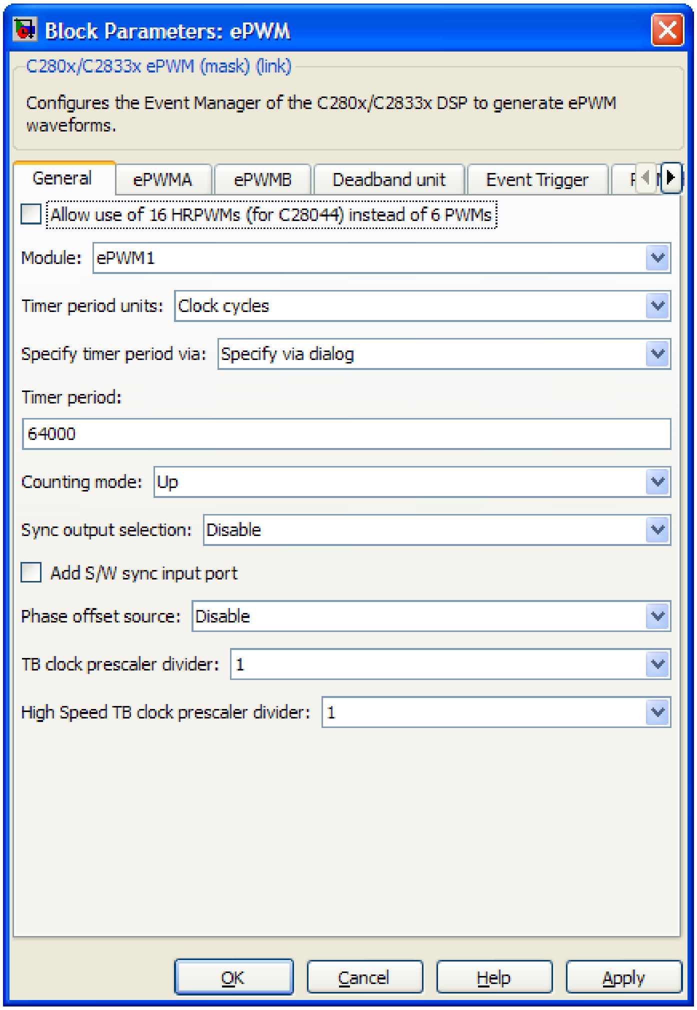
ePWM

Configures the Event Manager of the C280x/C2802x/C2803x/C28x3x DSP to generate ePWM waveforms. These DSPs contain multiple ePWM modules. Each module has two outputs, ePWMA and ePWMB. You can use the ePWM block to configure up to six ePWM modules.

When you enable the High-Resolution Pulse Width Modulator (HRPWM), the ePWM block uses the Scale Factor Optimizing Software Version 5 library (SFO_TI_Build_V5.lib). SFO_TI_Build_V5.lib can "dynamically determine the number of MEP steps per SYSCLKOUT period." For more information, consult *TMS320x28xx, 28xxx High-Resolution Pulse Width Modulator (HRPWM) Reference Guide*, Literature Number SPRU924, available at the Texas Instruments Web site.

Dialog Box

General Pane



Allow use of 16 HRPWMs (for C28044) instead of 6 PWMs

Enable all 16 High-Resolution PWM modules (HRPWM) on the C28044 digital signal controller when the PWM resolution is too low.

For example, the Spectrum Digital eZdsp™ F28044 board has a system clock of 100 MHz (200-kHz switching). At these frequencies, conventional PWM resolution is too low—approximately 9 bits or 10 bits. By comparison, the HRPWM resolution for the same board is 14.8 bits.

All the C280x/C2802x/C2803x/C28x3x/c2834x ePWM blocks in your model become HRPWM blocks, Thus, when you enable this parameter:

- Use the HRPWM parameters under the ePWMA tab to make additional configuration changes.
- Most of the configuration parameters under the ePWMB tab are unavailable.
- Your model can contain up to 16 C280x/C2803x/C28x3x ePWM blocks, provided you configure each one for a separate module. (For example, **Module** is `ePWM1`, `ePWM2`, and so on.)

For processors other than the C28044, deselect (disable) **Allow use of 16 HRPWMs (for C28044) instead of 6 PWMs**. To enable HRPWM for other processors, first determine how many HRPWM modules are available. Consult the Texas Instruments documentation for your processor, and then use the HRPWM parameters under the ePWMA tab to enable and configure HRPWM.

For additional information about the C28044 and HRPWM, consult the “References” on page 5-99 section.

Module

Specify which target ePWM module to use.

Timer period units

Specify the units of the **Timer period** or **Timer initial period** as `Clock cycles` (the default) or `Seconds`. When **Timer period units** is `Seconds`, the software down-converts the **Timer period** or **Timer initial period**, a `double` for the period register to a `uint16`. For best performance, select `Clock cycles`. Doing so reduces calculations and rounding errors.

Note If you set **Timer period units** to `Seconds`, enable support for floating-point numbers. In the model window, select **Simulation > Configuration Parameters**. In the Configuration Parameters dialog box, select **Code Generation > Interface**. Under **Software Environment**, enable **floating-point numbers**.

Specify timer period via
Timer period source

Configure the source of the timer period value. Selecting `Specify via dialog` changes the following parameter to **Timer period**. Selecting `Input port` changes the following parameter to **Timer initial period** and creates a timer period input port, **T**, on the block.

Timer period

Set the period of the PWM waveform in clock cycles or in seconds, as determined by the **Timer period units** parameter. When you enable HRMWM, you can enter a high-precision floating point value. The Time-Base Period High Resolution Register (TBPRDHR) stores the high-resolution portion of the timer period value.

Note The term *clock cycles* refers to the Time-base Clock on the DSP. See the **TB clock prescaler divider** topic for an explanation of Time-base Clock speed calculations.

Timer initial period

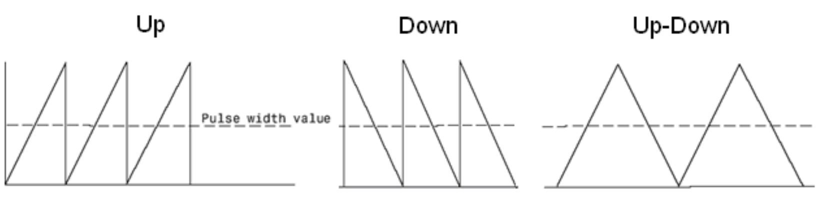
The period of the waveform from the time the PWM peripheral starts operation until the ePWM input port, **T**, receives a new value for the period. Use **Timer period units** to measure the period in clock cycles or in seconds.

Note The term *clock cycles* refers to the Time-base Clock on the DSP. See the **TB clock prescaler divider** topic for an explanation of Time-base Clock speed calculations.

Counting mode

Specify the counting mode in which to operate. This PWM module can operate in three distinct counting modes: `Up`, `Down`, and `Up-Down`. The `Down` option is not compatible with HRPWM. To avoid generating an error, do not select `Down` when you enable **HRPWM (Period)**.

The following illustration shows the waveforms that correspond to these three modes:



Sync output selection

This parameter corresponds to the SYNCOSEL field in the Time-Base Control Register (TBCTL).

Use this parameter to specify the event that generates a Time-base synchronization output signal, EPWMxSYNCO, from the Time-base (TB) submodule.

The available choices are:

- `EPWMxSYNCI or SWFSYNC` — a Synchronization input pulse or Software forced synchronization pulse, respectively. You can use this option to achieve precise synchronization across multiple ePWM modules by daisy chaining multiple Time-base (TB) submodules.
- `CTR=Zero` — Time-base counter equal to zero (TBCTR = 0x0000)
- `CTR=CMPB` — Time-base counter equal to counter-compare B (TBCTR = CMPB)
- `Disable` — Disable the EPWMxSYNCO output (the default)

Add S/W sync input port

Create an input port, **SYNC**, for a Time-base synchronization input signal, EPWMxSYNCI. You can use this option to achieve precise synchronization across multiple ePWM modules by daisy-chaining multiple Time-base (TB) submodules.

Enable DCAEVT1 sync

This parameter only appears in the C2802x and C2803x ePWM blocks.

Synchronize the ePWM time base to a DCAEVT1 digital compare event. Use this feature to synchronize this PWM module to the time base of another PWM module. Fine-tune the synchronization between the two modules using the **Phase offset value**. This option is not compatible with HRPWM. Enabling HRPWM disables this option.

Enable DCBEVT1 sync

This parameter only appears in the C2802x and C2803x ePWM blocks.

Synchronize the ePWM time base to a DCBEVT1 digital compare event. Use this feature to synchronize this PWM module to the time base of another PWM module. Fine-tune the synchronization between the two modules using the **Phase offset value**. This option is not compatible with HRPWM. Enabling HRPWM disables this option.

Phase offset source

Specify the source of a phase offset to apply to the Time-base synchronization input signal, EPWMxSYNCI from the **SYNC** input port. Selecting `Specify via dialog` creates the **Phase offset value** parameter. Selecting `Input port` creates a phase input port, **PHS**, on the block. Selecting `Disable`, the default value, prevents the application of phase offsets to the TB module.

Counting direction after phase synchronization

This parameter appears when **Counting Mode** is `Up-Down` and **Phase offset source** is `Specify via dialog` or `Input port`. Configure the timer to count up from zero, or down to zero, following synchronization. This parameter corresponds to the PHSDIR field of the Time-base Control Register (TBCTL).

Phase offset value

This field appears when you select `Specify via dialog` in **Phase offset source**.

Configure the phase offset (delay) between the following events:

- The arrival of the Time-base synchronization input signal (EPWMxSYNCI) on the **SYNC** input port
- The moment the Time-base (TB) submodule synchronizes the ePWM module.

Note Enter the **Phase offset value** in TBCLK cycles, from 0 to 65535. Do not use fractional seconds.

This parameter corresponds to the Time-Base Phase Register (TBPHS).

TB clock prescaler divider

Use the **TB clock prescaler divider** (CLKDIV) and the **High Speed TB clock prescaler divider** (HSPCLKDIV) to configure the Time-base clock speed (TBCLK) for the ePWM module. Calculate TBCLK using the following equation:

TBCLK = SYSCLKOUT/(HSPCLKDIV * CLKDIV)

For example, the default values of both CLKDIV and HSPCLKDIV are 1, and the default frequency of SYSCLKOUT is 100 MHz, so:

TBCLK = 100 MHz = 100 MHz/(1 * 1)

The choices for the **TB clock prescaler divider** are: 1, 2, 4, 8, 16, 32, 64, and 128.

The **TB clock prescaler divider** parameter corresponds to the CLKDIV field of the Time-base Control Register (TBCTL).

Note The frequency of SYSCLKOUT depends on the oscillator frequency and the configuration of PLL-based clock module. Changing the values of the PLL Control Register (PLLCR) affects the timing of all ePWM modules.

For more information, consult the “PLL-Based Clock Module” section of the data manual for your specific target (see “References” on page 5-99).

High Speed TB clock prescaler divider

See the **TB clock prescaler divider** topic for an explanation of the role of this value in setting the speed of the Time-base Clock. Choices are to divide by 1, 2, 4, 6, 8, 10, 12, and 14. Selecting **Enable HRPWM (Period)** forces this option to 1.

This parameter corresponds to the HSPCLKDIV field of the Time-base Control Register (TBCTL).

Enable swap module A and B

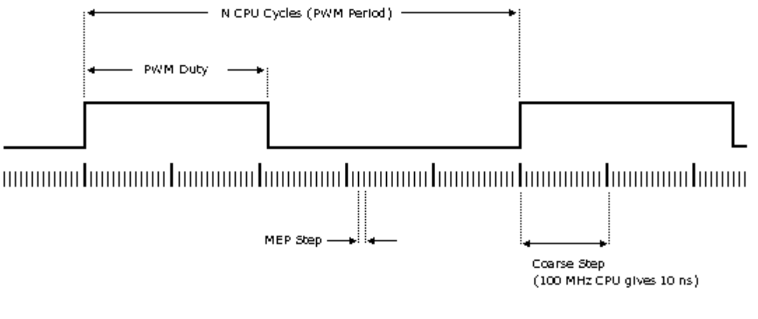
This parameter only appears in the C2802x and C2803x ePWM blocks.

Swap the ePWMA and ePWMB outputs. This option outputs the ePWMA signals on the ePWMB outputs and the ePWMB signals on the ePWMA outputs.

Enable HRPWM (Period)

This parameter only appears in the C2802x and C2803x ePWM blocks.

When the effective resolution for conventionally generated PWM is insufficient, consider using High Resolution PWM (HRPWM). The resolution of PWM is normally dependent upon the PWM frequency and the underlying system clock frequency. To address this limitation, HRPWM uses **Micro Edge Positioner (MEP)** ™ technology to position edges more finely by dividing each coarse system clock. The accuracy of the subdivision is on the order of 150ps. The following figure shows the relationship between one system clock and edge position in terms of **MEP** steps:



Enable HRPWM mode and control it via the Extension Register for HRPWM Period (TBPRDHR) register. When you enable this parameter, you can enter an 8–bit floating point value in for the **Timer period** parameter. This parameter enables the **Enable HRPWM (CMP)** option, and displays the **HRPWM loading mode**, **HRPWM control mode**, and **HRPWM edge control** mode options. Also configure **HRPWM control mode**.

Selecting Enable HRPWM (Period) forces **TB clock prescaler divider** and **High Speed TB clock prescaler divider** to 1. These settings match the HRPWM time base clock with the SYSCLKOUT frequency.

Enable HRPWM (CMP)

This parameter only appears in the C2802x and C2803x ePWM blocks.

Enable HRPWM mode and control it via the Extension Register for HRPWM Duty (CMPAHR) register. Also configure **HRPWM control mode**.

HRPWM loading mode

Determine when to transfer the value of the CMPAHR shadow to the active register:

- CTR=ZERO: Transfer the value when the time base counter equals zero (TBCTR = 0x0000).
- CTR=PRD: Transfer the value when the time base counter equals the period (TBCTR = TBPRD).
- CTR=Zero or CTR=PRD Transfer the value when either case is true.

This option configures the HRLOAD "Shadow Mode Bit" in the HRPWM Configuration Register (HRCNFG).

HRPWM control mode

Select which register controls the Micro Edge Positioner (MEP) step size. The **HRPWM control mode** option configures the CTLMODE "Control Mode Bits".

- `Duty control mode` uses the Extension Register for HRPWM Duty (CMPAHR) or the Extension Register for HRPWM Period (TBPRDHR) to control the MEP edge position.
- Select `Phase control mode` to use the Time Base Period High-Resolution Register (TBPRDHR) to control the MEP edge position.

The **HRPWM control mode** option configures the CTLMODE "Control Mode Bits" in the HRPWM Configuration Register (HRCNFG).

HRPWM edge control mode

Swap the ePWMA and ePWMB outputs. This parameter sets the SWAPAB field in the HRPWM Configuration Register (HRCNFG).

Use scale factor optimizer (SFO) software

Enable scale factor optimizing (SFO) software with HRPWM. This software dynamically determines the appropriate scaling factor for the Micro Edge Positioner (MEP) step size. The step size varies depending on operating conditions such as temperature and voltage. The SFO software reduces variability due to these conditions. For more information, see the "Scale Factor

Optimizing Software (SFO)" section of the *TMS320x2802x, 2803x Piccolo High Resolution Pulse Width Modulator (HRPWM) Reference Guide*, Literature Number: SPRUGE8.

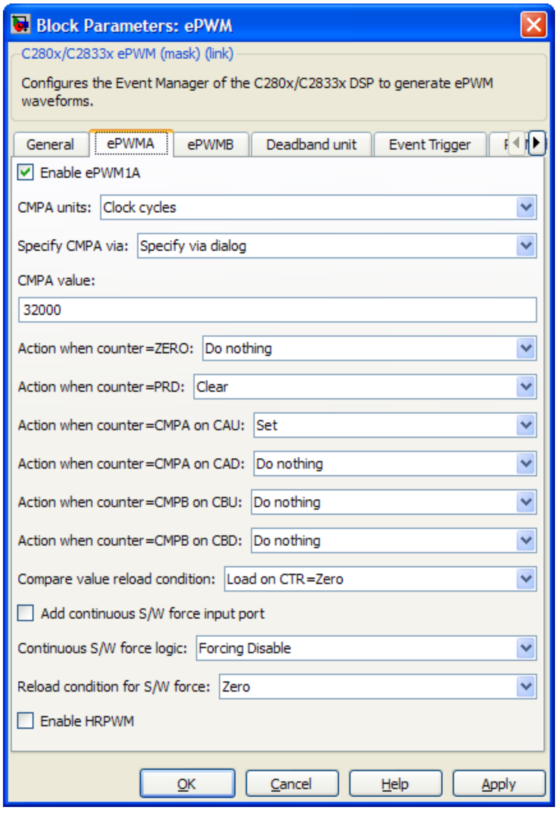
Enable auto convert

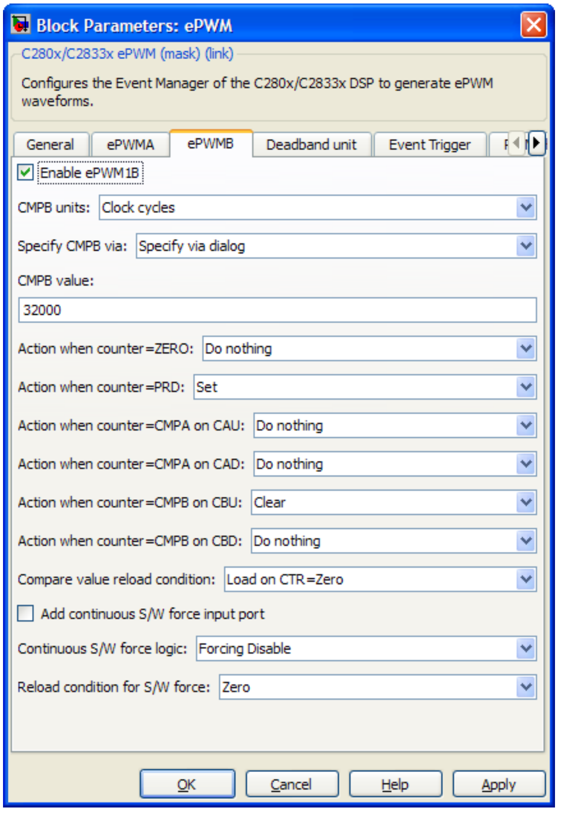
This parameter only appears in the C2802x and C2803x ePWM blocks.

Apply the scaling factor calculated by the SFO software to the controlling period or duty cycle. (Use the **HRPWM control mode** to select controlling period or duty cycle.) This parameter sets the AUTOCONV field in the HRPWM Configuration Register (HRCNFG).

ePWMA and ePWMB panes

Each ePWM module has two outputs, ePWMA and ePWMB. The **ePWMA output** pane and **ePWMB output** pane include the same settings, although the default values vary in some cases, as noted.


Block Parameters: ePWM
C280x/C2833x ePWM (mask) (link)
Configures the Event Manager of the C280x/C2833x DSP to generate ePWM waveforms.
General
ePWMA
ePWMB
Deadband unit
Event Trigger
Enable ePWM1A
CMPA units:
Clock cycles
Specify CMPA via:
Specify via dialog
CMPA value:
32000
Action when counter=ZERO:
Do nothing
Action when counter=PRD:
Clear
Action when counter=CMPA on CAU:
Set
Action when counter=CMPA on CAD:
Do nothing
Action when counter=CMPB on CBU:
Do nothing
Action when counter=CMPB on CBD:
Do nothing
Compare value reload condition:
Load on CTR=Zero
Add continuous S/W force input port
Continuous S/W force logic:
Forcing Disable
Reload condition for S/W force:
Zero
Enable HRPWM
OK
Cancel
Help
Apply



Enable ePWMxA
Enable ePWMxB

Enables the ePWMA and/or ePWMB output signals for the ePWM module identified on the **General** pane. By default, **Enable ePWMxA** is enabled, and **Enable ePWMxB** is disabled.

Note To **Enable ePWMxA** or **Enable ePWMxB**, also enable support for floating-point numbers: In the model window, select **Tools > Code Generation > Options**. In the Configuration Parameters dialog box, select **Code Generation > Interface**. Under **Software Environment**, enable **floating-point numbers**.

CMPA units
CMPB units

Specify the units used by the compare register: `Percentages` (the default) or `Clock cycles`.

Notes

- The term *clock cycles* refers to the Time-base Clock on the DSP. See the **TB clock prescaler divider** topic for an explanation of Time-base Clock speed calculations.
- Percentages use additional computation time in generated code and can decrease performance.
- If you set **CMPA units** or **CMPB units** to `Percentages`, also enable support for floating-point numbers: In the model window, select **Simulation > Configuration Parameters**. In the Configuration Parameters dialog box, select **Code Generation > Interface**. Under **Software Environment**, enable **floating-point numbers**.

Specify CMPA via
Specify CMPB via

Specify the source of the pulse width. If you select `Specify via dialog` (the default), enter a value in the **CMPA value** or **CMPB value** field. If you select `Input port`, set the value using an input port, **WA** or **WB**, on the block. If you select `Input port` also set **CMPA initial value** or **CMPB initial value**.

CMPA value
CMPB value

This field appears when you choose `Specify via dialog` in **CMPA source** or **CMPB source**. Enter a value that specifies the pulse width, in the units specified in **CMPA units** or **CMPB units**.

CMPA initial value
CMPB initial value

This field appears when you set **CMPA source** or **CMPB source** to `Input port`. Enter the initial pulse width of CMPA or CMPB the PWM peripheral uses when it starts operation. Subsequent inputs to the **WA** or **WB** ports change the CMPA or CMPB pulse width.

Action when counter=ZERO
Action when counter=PRD
Action when counter=CMPA on CAU
Action when counter=CMPA on CAD
Action when counter=CMPB on CBU
Action when counter=CMPB on CBD

These settings, along with the other remaining settings in the **ePWMA output** and **ePWMB output** panes, determine the behavior of the Action Qualifier (AQ) submodule. The AQ module determines which events are converted into various action types, producing the required switched waveforms of the ePWMxA and ePWMxB output signals.

For each of these four fields, the available choices are `Do nothing`, `Clear`, `Set`, and `Toggle`.

The default values for these fields vary between the **ePWMA output** and **ePWMB output** panes.

The following table shows the defaults for each of these panes when you set **Counting mode** to `Up` or `Up-Down`:

Action when counter =...	ePWMA output pane	ePWMB output pane
ZERO	Do nothing	Do nothing
PRD	Clear	Set
CMPA on CAU	Set	Do nothing
CMPA on CAD	Do nothing	Do nothing
CMPB on CBU	Do nothing	Clear
CMPB on CBD	Do nothing	Do nothing

The following table shows the defaults for each of these panes when you set **Counting mode** to `Down`:

Action when counter =...	ePWMA output pane	ePWMB output pane
ZERO	Do nothing	Do nothing
PRD	Clear	Set
CMPA on CAD	Do nothing	Do nothing
CMPB on CBD	Do nothing	Do nothing

For a detailed discussion of the AQ submodule, consult the *TMS320x280x Enhanced Pulse Width Modulator (ePWM) Module Reference Guide* (SPRU791), available on the Texas Instruments Web site.

Compare value reload condition
Add continuous S/W force input port
Continuous S/W force logic
Reload condition for S/W force

These four settings determine how the action-qualifier (AQ) submodule handles the S/W force event, an asynchronous event initiated by software (CPU) via control register bits.

Compare value reload condition determines if and when to reload the Action-qualifier S/W Force Register from a shadow register. Choices are `Load on CTR=Zero` (the default), `Load on CTR=PRD`, `Load on either`, and `Freeze`.

Add continuous S/W force input port creates an input port, **SFA**, which you can use to control the software force logic. Send one of the following values to **SFA** as an unsigned integer data type:

- 0 = `Forcing Disable`: Do nothing. The default.
- 1 = `Forcing Low`: Clear low
- 2 = `Forcing High`: Set high

If you did not create the **SFA** input port, you can use **Continuous S/W force logic** to select which type of software force logic to apply. The choices are:

- `Forcing Disable`: Do nothing. The default.
- `Forcing Low`: Clear low
- `Forcing High`: Set high

Reload condition for S/W force — Choices are `Zero` (the default), `Period`, `Either period or zero`, and `Immediate`.

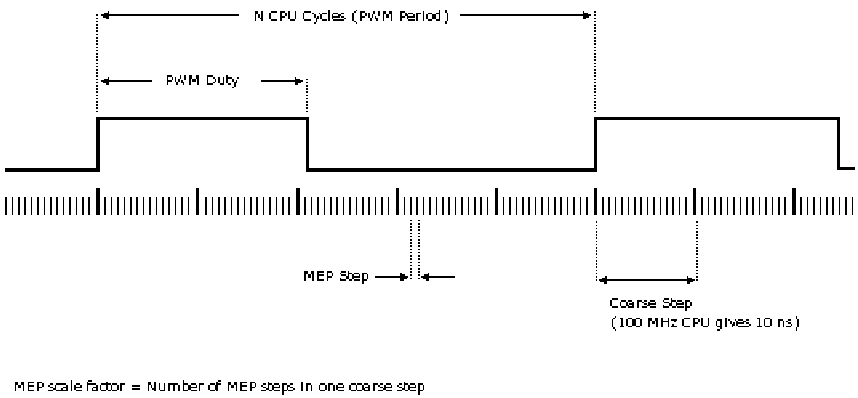
Inverted version of ePWMxA

Only the ePWMB pane on the C2802x and C2803x blocks displays this option. Invert the ePWMxA signal and output it on the ePWMxB outputs. This parameter sets the SELOUTB field in the HRPWM Configuration Register (HRCNFG).

Enable HRPWM

This parameter appears at this position in the C280x and C2833x ePWM blocks.

Select to enable High Resolution PWM settings. When the effective resolution for conventionally generated PWM is insufficient, consider High Resolution PWM (HRPWM). The resolution of PWM is normally dependent upon the PWM frequency and the underlying system clock frequency. To address this limitation, HRPWM uses **Micro Edge Positioner (MEP)** technology to position edges more finely by dividing each coarse system clock. The accuracy of the subdivision is on the order of `150ps`. The following figure shows the relationship between one system clock and edge position in terms of **MEP** steps:



HRPWM loading mode

This parameter appears at this position in the C280x and C2833x ePWM blocks.

Determine when to transfer the value of the CMPAHR shadow to the active register:

- CTR=ZERO: Transfer the value when the time base counter equals zero (TBCTR = 0x0000).
- CTR=PRD: Transfer the value when the time base counter equals the period (TBCTR = TBPRD).
- CTR=Zero or CTR=PRD Transfer the value when either case is true.

HRPWM control mode

This parameter appears at this position in the C280x and C2833x ePWM blocks.

Select which register controls the Micro Edge Positioner (MEP) step size. The **HRPWM control mode** option configures the CTLMODE "Control Mode Bits".

- `Duty control mode` uses the Extension Register for HRPWM Duty (CMPAHR) or the Extension Register for HRPWM Period (TBPRDHR) to control the MEP edge position.
- Select `Phase control mode` to use the Time Base Period High-Resolution Register (TBPRDHR) to control the MEP edge position.

The **HRPWM control mode** option configures the CTLMODE "Control Mode Bits" in the HRPWM Configuration Register (HRCNFG).

HRPWM edge control mode

This parameter appears at this position in the C280x and C2833x ePWM blocks.

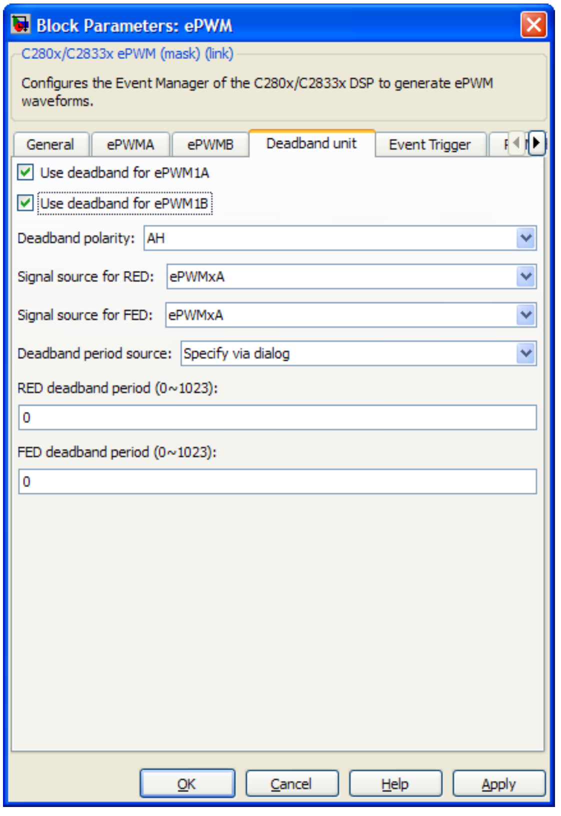
Swap the ePWMA and ePWMB outputs. This parameter sets the SWAPAB field in the HRPWM Configuration Register (HRCNFG).

Use scale factor optimizer (SFO) software

Enable scale factor optimizing (SFO) software with HRPWM. This software dynamically determines the appropriate scaling factor for the Micro Edge Positioner (MEP) step size. The step size varies depending on operating conditions such as temperature and voltage. The SFO software reduces variability due to these conditions. For more information, see the “Scale Factor Optimizing Software (SFO)” section of the *TMS320x2802x, 2803x Piccolo High Resolution Pulse Width Modulator (HRPWM) Reference Guide*, Literature Number: SPRUGE8.

Deadband Unit Pane

The **Deadband unit** pane lets you specify parameters for the Dead-Band Generator (DB) submodule.



Use deadband for ePWMxA
Use deadband for ePWMxB

Enables a deadband area of no signal overlap between pairs of ePWM output signals. This check box is cleared by default.

Enable half-cycle clocking

This parameter only appears in the C2802x and C2803x ePWM blocks.

To double the deadband resolution, enable half-cycle clocking. This option clocks the deadband counters at TBCLK*2. When you disable this option, the deadband counters use full-cycle clocking (TBCLK*1).

Deadband polarity
Configure the deadband polarity as `AH` (active high, the default), `AL` (active low), `AHC` (active high complementary), or `ALC` (active low complementary).

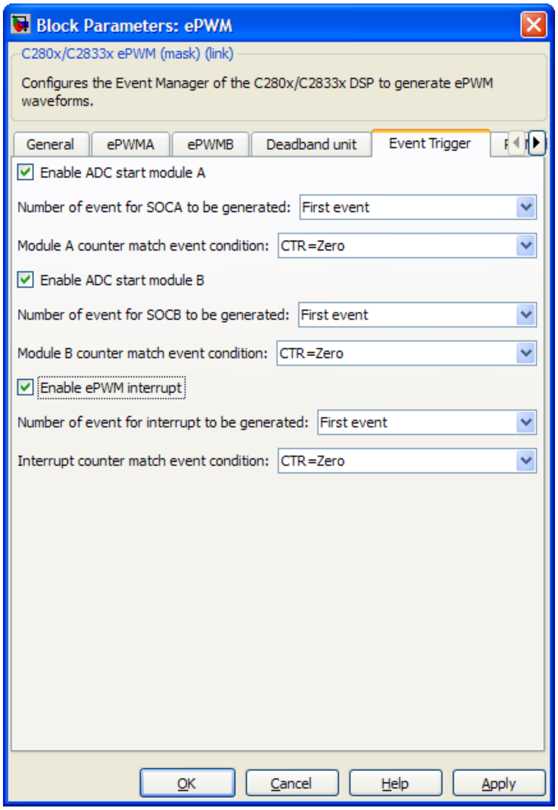
Deadband period source
Specify the source of the control logic. Choose `Specify via dialog` (the default) to enter explicit values, or `Input port` to use a value from the input port.

RED deadband period
This field appears only when you select **Use deadband for ePWMxA** in the **ePWMA output** pane. Enter a value from `0` to `1023` to specify a rising edge delay.

FED deadband period
This field appears only when you select **Use deadband for ePWMxB** in the **ePWMB output** pane. Enter a value from `0` to `1023` to specify a falling edge delay.

Event Trigger Pane

Configure ADC Start of Conversion (SOC) by one or both of the ePWMA and ePWMB outputs.



Enable ADC start module A

When you select this option, ePWM starts the Analog-to-Digital Conversion (ADC) for module A. By default, the software clears (disables) this option.

Number of event for SOCA to be generated

When you select **Enable ADC start module A**, this field specifies the number of the event that triggers ADC Start of Conversion for Module A (SOCA): `First event` triggers ADC start of conversion with every event (the default). `Second event` triggers ADC start of conversion with every second event. `Third event` triggers ADC start of conversion with every third event.

Module A counter match event condition

When you select **Enable ADC start module A**, this field specifies the counter match condition that triggers an ADC start of conversion event. The choices are:

DCAEVT1 soc and DCBEVT1 soc

(For C2802x and C2803x only) When the ePWM asserts a DCAEVT1 or DCBEVT1 digital compare event. Use this feature to synchronize this PWM module to the time base of another PWM module. Fine-tune the synchronization between the two modules using the **Phase offset value**.

CTR=Zero

When the ePWM counter reaches zero (the default).

CTR=PRD

When the ePWM counter reaches the period value.

CTR=Zero or CTR=PRD

When the time base counter equals zero (TBCTR = 0x0000) or when the time base counter equals the period (TBCTR = TBPRD).

CTRU=CMPA

When the ePWM counter reaches the compare A value on the way up.

CTRD=CMPA

When the ePWM counter reaches the compare A value on the way down.

CTRU=CMPB

When the ePWM counter reaches the compare B value on the way up.

CTRD=CMPB

When the ePWM counter reaches the compare B value on the way down.

Enable ADC start module B

When you select this option, ePWM starts the Analog-to-Digital Conversion (ADC) for module B. By default, the software clears (disables) this option.

Number of event for SOCB to be generated

When you select **Enable ADC start module B**, this field specifies the number of the event that triggers ADC start of conversion: `First event` triggers ADC start of conversion with every event (the default), `Second event` triggers ADC start of conversion with every second event, and `Third event` triggers ADC start of conversion with every third event.

Module B counter match event condition

When you select **Enable ADC start module B**, this field specifies the counter match condition that triggers an ADC start of conversion event. The choices are the same as for **Module A counter match event condition**.

Enable ePWM interrupt

Select this option to generate interrupts based on different events defined by **Number of event for interrupt to be generated** and **Interrupt counter match event condition**. By default, the software clears (disables) this option.

Number of event for interrupt to be generated

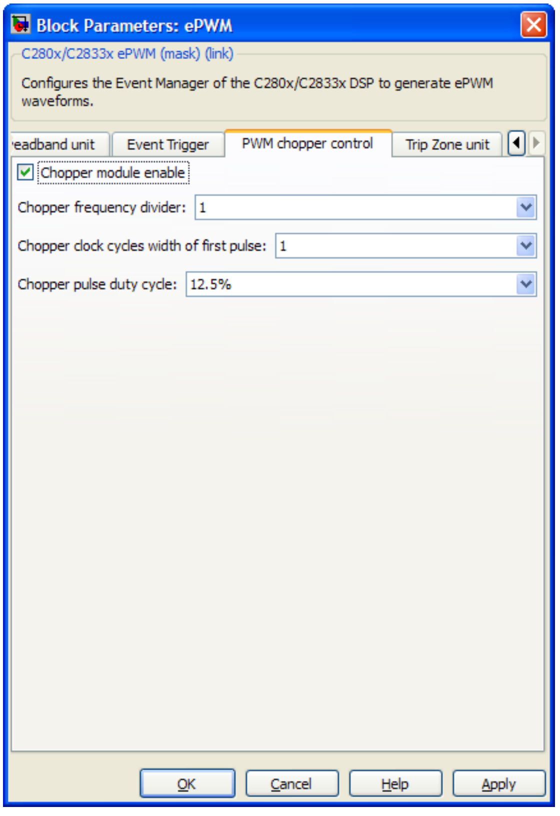
When you select **Enable ePWM interrupt**, this field specifies the number of the event that triggers the ePWM interrupt: `First event` triggers ePWM interrupt with every event (the default), `Second event` triggers ePWM interrupt with every second event, and `Third event` triggers ePWM interrupt with every third event.

Interrupt counter match event condition

When you select **Enable ePWM interrupt**, this field specifies the counter match condition that triggers ePWM interrupt. The choices are the same as for **Module A counter match event condition**.

PWM Chopper Control Pane

The **PWM chopper control** pane lets you specify parameters for the PWM-Chopper (PC) submodule. The PC submodule uses a high-frequency carrier signal to modulate the PWM waveform generated by the AQ and DB modules.



Chopper module enable

Select to enable the chopper module. Use of the chopper module is optional, so this check box is cleared by default.

Chopper frequency divider

Set the prescaler value that determines the frequency of the chopper clock. The system clock speed is divided by this value to determine the chopper clock frequency. Choose an integer value from 1 to 8.

Chopper clock cycles width of first pulse

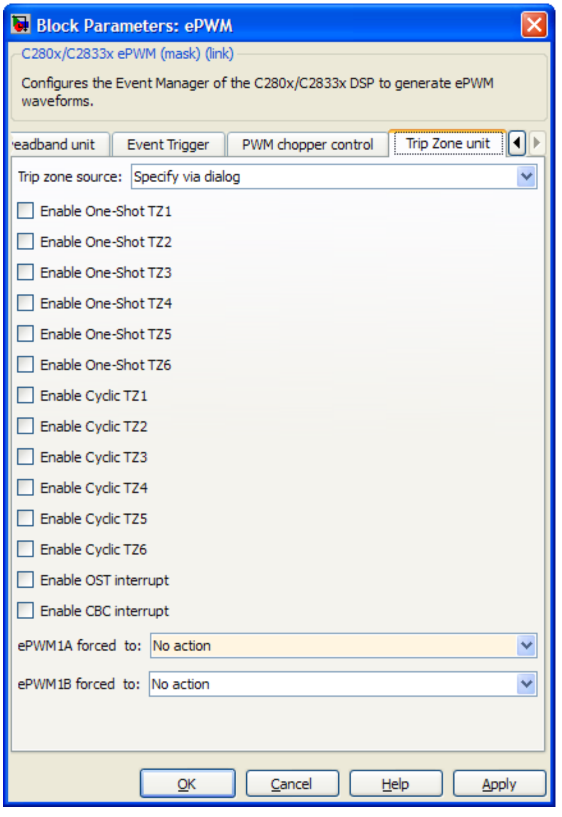
Choose an integer value from 1 to 16 to set the width of the first pulse. This feature provides a high-energy first pulse for a hard and fast power switch turn on.

Chopper pulse duty cycle

The duty cycles of the second and subsequent pulses are also programmable. Choices are 12.5%, 25%, 37.5%, 50%, 62.5%, 75%, and 87.5%.

Trip Zone Unit Pane

The **Trip Zone unit** pane lets you specify parameters for the Trip-zone (TZ) submodule. Each ePWM module receives six TZ signals (TZ1 to TZ6) from the GPIO MUX. These signals indicate external fault or trip conditions. Use the settings in this pane to program the EPWM outputs to respond when faults occur.



Trip zone source

Specify the source of the control logic to enable or disable the TZ Interrupts (**One shot TZ1-TZ6** and **Cyclic TZ1-TZ6**). Select `Specify via dialog` (the default) to enable specific Trip-zone signals in the block dialog. Choose `Input port` to enable specific Trip-zone signals using a block input port, **TZSEL**.

If you select `Input port`, use the following bit operation to determine the value of the 16-bit integer to send to the **TZSEL** input port:

$TZSEL\ INPUT\ VALUE = (OSHT6*2^{13} + OSHT5*2^{12} + OSHT4*2^{11} + OSHT3*2^{10} + OSHT2*2^{9} + OSHT1*2^{8} + CBC6*2^{5} + CBC5*2^{4} + CBC4*2^{3} + CBC3*2^{2} + CBC2*2^{1} + CBC1*2^{0})$

The software uses the higher 8 bits for the **One shot TZ1-TZ6** and the lower 8 bits for **Cyclic TZ1-TZ6**. You can set up a group of TZ sources (1~6), use a bit operation to combine them into an integer, and then feed the integer to TZSEL.

For example, to enable One Shot TZ6 (OSHT6) and One Shot TZ5 (OSHT5) as trip zone sources, set OSHT6 and OSHT5 to “1” and leave the remaining values as “0”.

TZSEL INPUT VALUE = $(1*2^{13} + 1*2^{12} + 0*2^{11} \ldots)$

TZSEL INPUT VALUE = (8192 + 4096 + 0 …)

TZSEL INPUT VALUE = 12288

When the block receives this value, it applies it to the TZSEL register as a binary value: `11000000000000`.

For more information, see the ”Trip-Zone Submodule Control and Status Registers” section of the *TMS320x28xx, 28xxx Enhanced Pulse Width Modulator (ePWM) Module Reference Guide*, Literature Number: SPRU791 on www.ti.com

Enable One-Shot TZ1
Enable One-Shot TZ2
Enable One-Shot TZ3
Enable One-Shot TZ4
Enable One-Shot TZ5
Enable One-Shot TZ6

Select any of these check boxes to enable the corresponding Trip-zone signal in One-Shot Mode. In this mode, when the trip event is active, the software performs the corresponding action on the EPWMxA/B output immediately and latches the condition. You can unlatch the condition using software control.

Enable Cyclic TZ1
Enable Cyclic TZ2
Enable Cyclic TZ3
Enable Cyclic TZ4
Enable Cyclic TZ5
Enable Cyclic TZ6

Select any of these check boxes to enable the corresponding Trip-zone signal in Cycle-by-Cycle Mode. In this mode, when the trip event is active, the software performs the corresponding action on the EPWMxA/B output immediately and latches the condition. In Cycle-by-Cycle Mode, the software automatically clears condition when the PWM Counter reaches zero. Therefore, in Cycle-by-Cycle Mode, every PWM cycle resets or clears the trip event.

Enable OST Interrupt

Generate an interrupt when the one shot (OST) triggering event occurs.

Enable CBC Interrupt

Generate an interrupt when the cyclic or cycle-by-cycle (CBC) triggering event occurs.

ePWMxA forced to
ePWMxB forced to

Upon a fault condition, the software overrides and forces the ePWMxA and/or ePWMxB outputs to one of the following states: `No action` (the default), `High`, `Low`, or `Hi-Z (High Impedance)`.

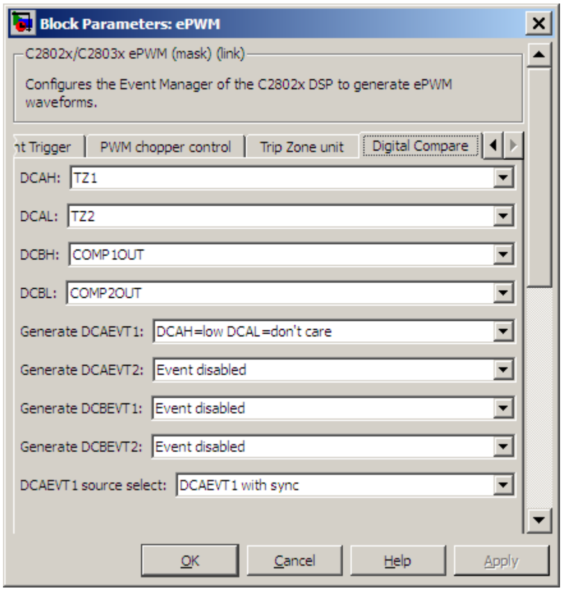
Digital Compare

Use the **Digital Compare** pane to configure the Digital Compare (DC) submodule.

Each digital compare (DC) submodule receives three TZ signals (TZ1 to TZ3) from the GPIO MUX, and three COMP signals from the COMP. These signals indicate fault or trip conditions that are external to the

PWM submodule. Use the settings in this pane to output specific DC events in response to those external signals. These DC events feed directly into the Time-base, Trip-zone, and Event-trigger submodules.

For more information, see the "Digital Compare (DC) Submodule" section of the *TMS320x2802x, 2803x Piccolo Enhanced Pulse Width Modulator (ePWM) Module Reference Guide*, Literature Number: SPRUGE9.



DCAH, **DCBH**
: If the TZ or COMP event you select occurs, assert a high signal. Qualify this signal using the **Generate DCAEVT#**, **Generate DCBEVT#** options.

DCAL, **DCBL**
: If the TZ or COMP event you select occurs, assert a low signal. Qualify this signal using the **Generate DCAEVT#**, **Generate DCBEVT#** options.

Generate DCAEVT#, **Generate DCBEVT#**

Qualify the signals that generate DC events, such as DCAEVT# or DCBEVT#. Select the states of **DCAH**, **DCBH**, **DCAL**, and **DCBL** that generate the event. To disable this feature, choose the `Event disabled` option.

DCAEVT# source select, **DCBEVT# source select**

This parameter controls two separate aspects of triggering DC events:

- Triggering filtered or unfiltered DC event. (Configures DCACTL[EVT1SRCSEL] or DCACTL[EVT2SRCSEL].)
- Trigger the DC event synchronously or asynchronously. (Configures DCACTL[EVT1FRCSYNCSEL] or DCACTL[EVT2FRCSYNCSEL].)

Filtering

- Options that begin with `DCAEVT#` or `DCAEVT#` do not apply filtering to DC events. Qualified signals trigger DC events.
- Options that begin with `DCEVTFILT` apply filtering to DC events. Qualified signals pass through filtering circuits before triggering DC events. This filtering is not configurable in the ePWM block. For more information, refer to the "Event Filtering" section of the *TMS320x2802x, 2803x Piccolo Enhanced Pulse Width Modulator (ePWM) Module Reference Guide*, Literature Number: SPRUGE9.

Synchronizing

- Options that end with `async` trigger DC events asynchronously. When the qualified or filtered signals exist, the DC submodule triggers the DC event immediately.
- Options that end with `sync` trigger DC events synchronously. Once the qualified or filtered signals exist, the DC submodule triggers the DC event in sync with the TBCLK signal.

References

For more information, consult the following references, available at the Texas Instruments Web site:

- *TMS320x28xx, 28xxx Enhanced Pulse Width Modulator (ePWM) Module Reference Guide*, literature number SPRU791
- *TMS320x280x, 2801x, 2804x High Resolution Pulse Width Modulator Reference Guide*, literature number SPRU924E
- *TMS320x2802x, 2803x Piccolo Enhanced Pulse Width Modulator (ePWM) Module Reference Guide*, literature number SPRUGE9
- *TMS320x2802x, 2803x Piccolo High Resolution Pulse Width Modulator (HRPWM) Reference Guide*, literature number SPRUGE8
- *Using the ePWM Module for 0% - 100% Duty Cycle Control Application Report*, literature number SPRU791
- *Configuring Source of Multiple ePWM Trip-Zone Events*, literature number SPRAAR4
- *TMS320F2809, TMS320F2808, TMS320F2806 TMS320F2802, TMS320F2801 TMS320C2802, TMS320C2801, and TMS320F2801x DSPs Data Manual*, literature number SPRS230
- *TMS320F28044 Digital Signal Processor Data Manual*, literature number SPRS357
- *TMS320F28335/28334/28332 TMS320F28235/28234/28232 Digital Signal Controllers (DSCs) Data Manual*, literature number SPRS439

See Also

C280x/C28x3x ADC

"ePWM" on page 5-891

C280x/C2803x/C28x3x/c2834x eQEP

Purpose

Quadrature encoder pulse circuit

Library

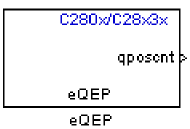
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

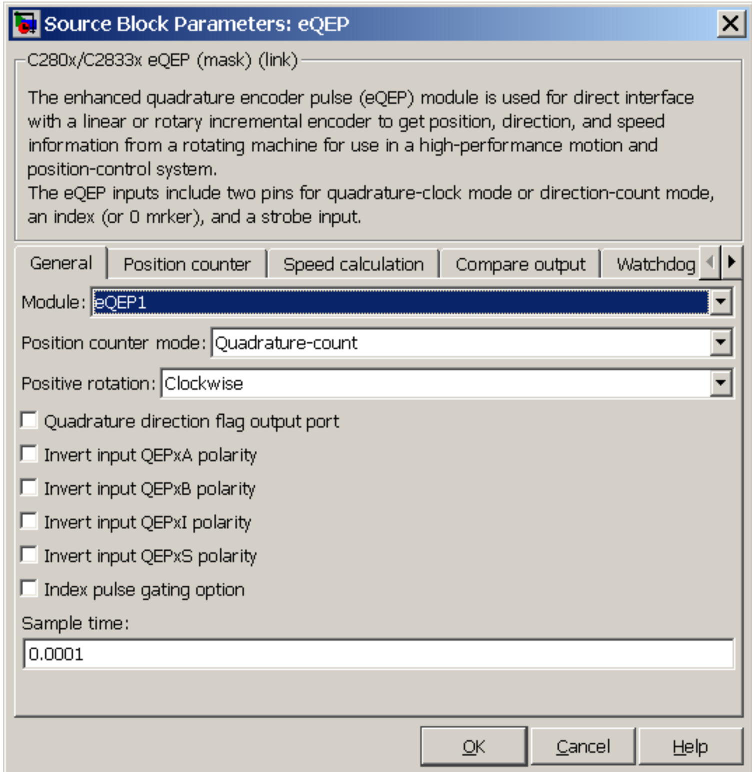
Description



The enhanced quadrature encoder pulse (eQEP) module is used for direct interface with a linear or rotary incremental encoder to get position, direction, and speed information from a rotating machine for use in a high-performance motion and position-control system.

Dialog Box

General Pane



Module

If more than one eQEP module is available on your processor, select the module this block configures.

Position counter mode

The input signals QEPA and QEPB are processed by the Quadrature Decoder Unit (QDU) to produce clock (QCLK) and direction (QDIR) signals. Choose the position counter mode appropriate to the way the input to the eQEP module is encoded.

Choices are `Quadrature-count` (the default), `Direction-count`, `Up-count`, and `Down-count`.

Positive rotation

This field appears only when you choose `Quadrature-count` in **Position counter mode**. Choose the direction that represents positive rotation: `Clockwise` (the default) or `Counterclockwise`.

External clock rate

This field appears only when you choose `Direction-count`, `Up-count`, or `Down-count` in **Position counter mode**. In these cases, you can program clock generation to the position counter to occur on both rising and falling edges of the QEPA input or on the rising edge only. The effect of choosing the former is increasing the measurement resolution by a factor of 2. Choices are `2x resolution:  Count the rising/falling edge` (the default) or `1x resolution:  Count the rising edge only`.

Quadrature direction flag output port

This check box appears only when you choose `Quadrature-count` in **Position counter mode**. Select this check box if you want to create a port on the block that gives access to the direction flag of the quadrature module.

Invert input QEPxA polarity
Invert input QEPxB polarity
Invert input QEPxI polarity
Invert input QEPxS polarity

Select any of these check boxes to invert the polarity of the respective eQEP input signal.

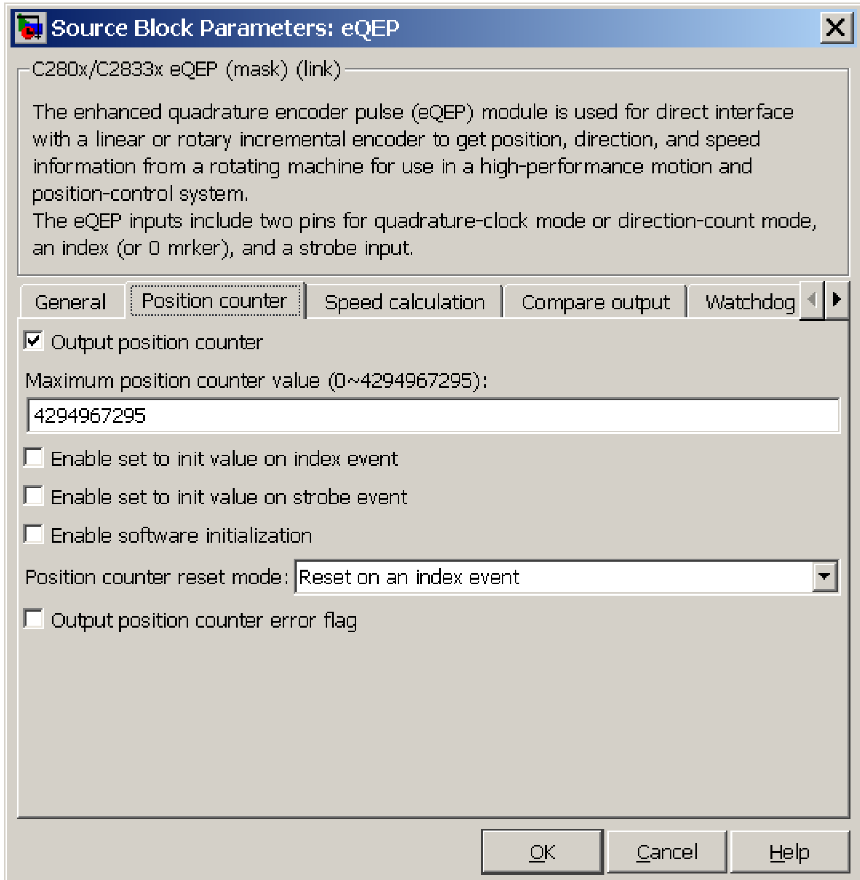
Index pulse gating option

Select this check box to enable gating of the index pulse.

Sample time

Enter the sample time in seconds.

Position Counter Pane



Output position counter

This check box is selected by default. Leave it selected to output the position counter signal PCSOUT from the position counter and control unit (PCCU).

Maximum position counter value
Enter a maximum value for the position counter. Enter a value from `0` to `4294967295`. The value defaults to the maximum allowed value of `4294967295`.

Enable set to init value on index event
Select to set the position counter to its initialization value on an index event. This check box is cleared by default.

Set to init value on index event
This field appears only when **Enable set to init value on index event** is selected. Choose to set the position counter to its initialization value on the `Rising edge` (the default) or the `Falling edge` of the index input.

Enable set to init value on strobe event
Select to set the position counter to its initialization value on a strobe event. This check box is cleared by default.

Set to init value on strobe event
This field appears only when **Enable set to init value on strobe event** is selected. `Rising edge`, the default option, sets the position counter to its initialization value on the rising edge of the strobe input. In the forward direction, `Depending on direction` sets the position counter to its initialization value on the rising edge of the strobe input. In the reverse direction, `Depending on direction` sets the position counter to its initialization value on the falling edge of the strobe input.

Enable software initialization
Select to allow the position counter to be set to its initialization value via software. This check box is cleared by default.

Software initialization source
This field appears only when **Enable software initialization** is selected. Choose `Set to init value at start up` (the default) or `Input port` to receive the control logic through the input port.

Initialization value

This field appears only when **Enable set to init value on index event**, **Enable set to init value on strobe event**, or **Enable software initialization** check box is selected. Enter the initialization value for the position counter. Enter a value from `0` to `4294967295`. The value defaults to `2147483648`.

Position counter reset mode

Choose a position counter reset mode, depending on the nature of the system the eQEP module is working with: `Reset on an index event` (the default), `Reset on the maximum position`, `Reset on the first index event`, or `Reset on a time unit event`.

Output position counter error flag

This check box appears only when **Position counter reset mode** is set to `Reset on an index event`. Select this check box to output the position counter error flag on error.

Output latch position counter on index event

This check box appears only when **Position counter reset mode** is set to `Reset on the maximum position` or `Reset on the first index event`. The eQEP index input can be configured to latch the position counter (QPOSCNT) into QPOSILAT on occurrence of a definite event on this pin. Select this check box to latch the position counter on each index event.

Index event latch of position counter

This field appears only when the **Output latch position counter on index event** check box is selected. Choose one of the following events to configure the eQEP position counter to latch on that event: `Rising edge`, `Falling edge`, or `Software index marker via input port`.

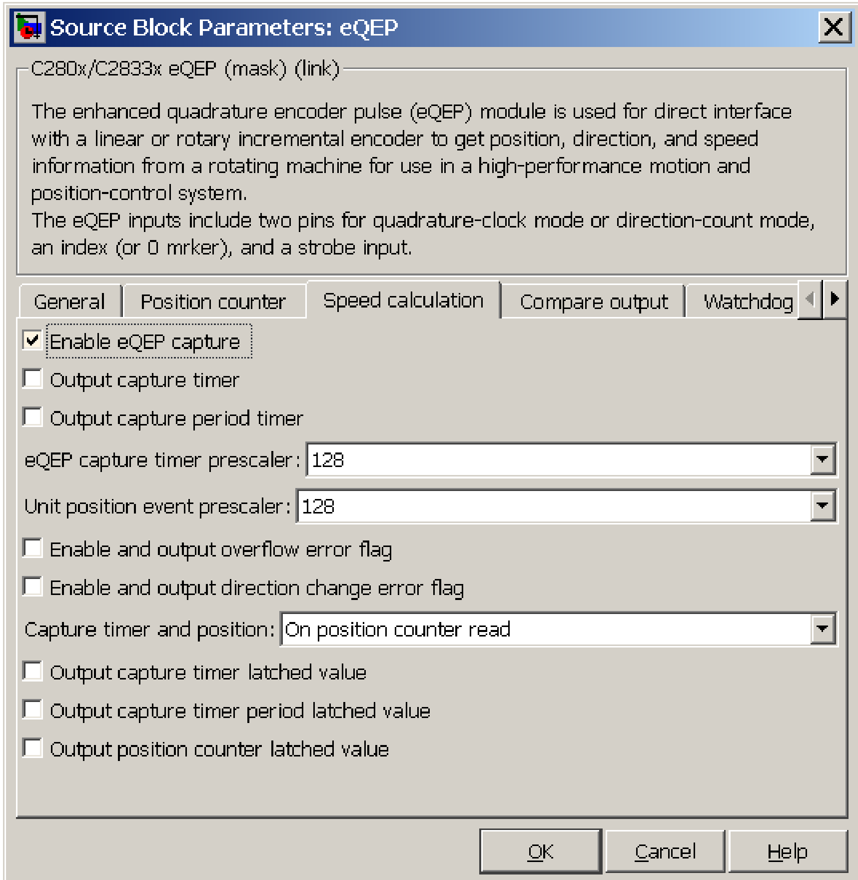
Output latch position counter on strobe event

This check box appears only when **Position counter reset mode** is set to `Reset on the maximum position` or `Reset on the first index event`. The eQEP strobe input can be configured to latch the position counter (QPOSCNT) into QPOSSLAT on occurrence of a definite event on this pin. Select this check box to latch the position counter on each strobe event.

Strobe event of latched position counter

This field appears only when the **Output latch position counter on strobe event** check box is selected. Choose `Rising edge` to latch on the rising edge of the strobe event input, or `Depending on direction` to latch on the rising edge in the forward direction and the falling edge in the reverse direction.

Speed Calculation Pane



Enable QEP capture

The eQEP peripheral includes an integrated edge capture unit to measure the elapsed time between the unit position events. Check this check box to enable the edge capture unit. This check box is cleared by default.

Output capture timer
Select this check box to output the capture timer into the capture period register. This check box is cleared by default.

Output capture period timer
Select this check box to output the capture period into the capture period register. This check box is cleared by default.

eQEP capture timer prescaler
The eQEP capture timer runs from prescaled SYSCLKOUT. The capture timer period is the value of SYSCLKOUT divided by the value you choose in this field. Choices are `1`, `2`, `4`, `8`, `16`, `32`, `64`, and `128` (the default).

Unit position event prescaler
The timing of the unit position event is determined by prescaling the quadrature-clock (QCLK). QCLK is divided by the value you choose in this popup. Choices are `4`, `8`, `16`, `32`, `64`, `128`, `256`, `512`, `1024`, and `2048` (the default).

Enable and output overflow error flag
Select this check box to enable and output the eQEP overflow error flag in the event of capture timer overflow between unit position events.

Enable and output direction change error flag
Select this check box to enable and output the direction change error flag.

Capture timer and position
Choose the event that triggers the latching of the capture timer and capture period register: `On position counter read` (the default) or `On unit time-out event`.

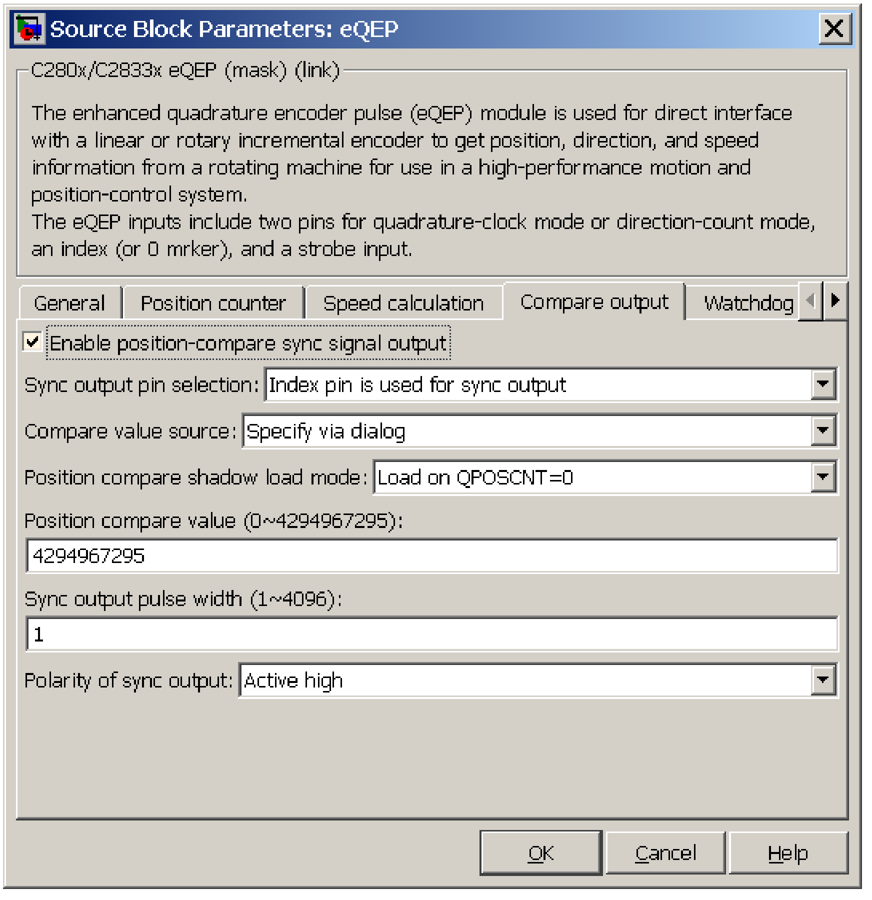
Unit timer period
This field appears only when you choose `On unit time-out event` in **Capture timer and position**. Enter a value for the unit timer period from `0` to `4294967295`. The value defaults to `100000000`.

Output capture timer latched value
Select this check box to output the capture timer latched value from the QCTMRLAT register.

Output capture timer period latched value
Select this check box to output the capture timer period latched value from the QCPRDLAT register.

Output position counter latched value
Select this check box to output the position counter latched value from the QPOSLAT register.

Compare Output Pane



Enable position-compare sync signal output

The eQEP peripheral includes a position-compare unit that is used to generate the position-compare sync signal on compare match between the position counter register (QPOSCNT) and the position-compare register (QPOSCMP). Select this check box to enable the position-compare sync signal output. This check box is cleared by default.

Sync output pin selection

Choose which pin is used for the sync signal output. Choices are `Index pin is used for sync output` (the default) and `Strobe pin is used for sync output`.

Compare value source

Choose the source of the value to use in the position comparison. Choose `Specify via dialog` (the default) to specify a fixed value or `Input port` to read the value from the input port.

Position compare shadow load mode

This field lets you enable or disable shadow mode for use in generating the position-compare sync signal (shadow mode is enabled by default). When shadow mode is enabled, you can also choose an event to trigger the loading of the shadow register value into the active register.

Choose `Disable shadow mode` to disable shadow mode. Choose `Load on QPOSCNT=0` (the default) to load on the position-counter zero event. Choose `Load on QPOSCNT=QPOSCMP` to load on compare match.

Position compare value

This field appears only when you choose `Specify via dialog` in **Compare value source**. Enter a value from `0` to `4294967295`. The value defaults to `4294967295`. This value is loaded into the position-compare register (QPOSCMP).

Sync output pulse width

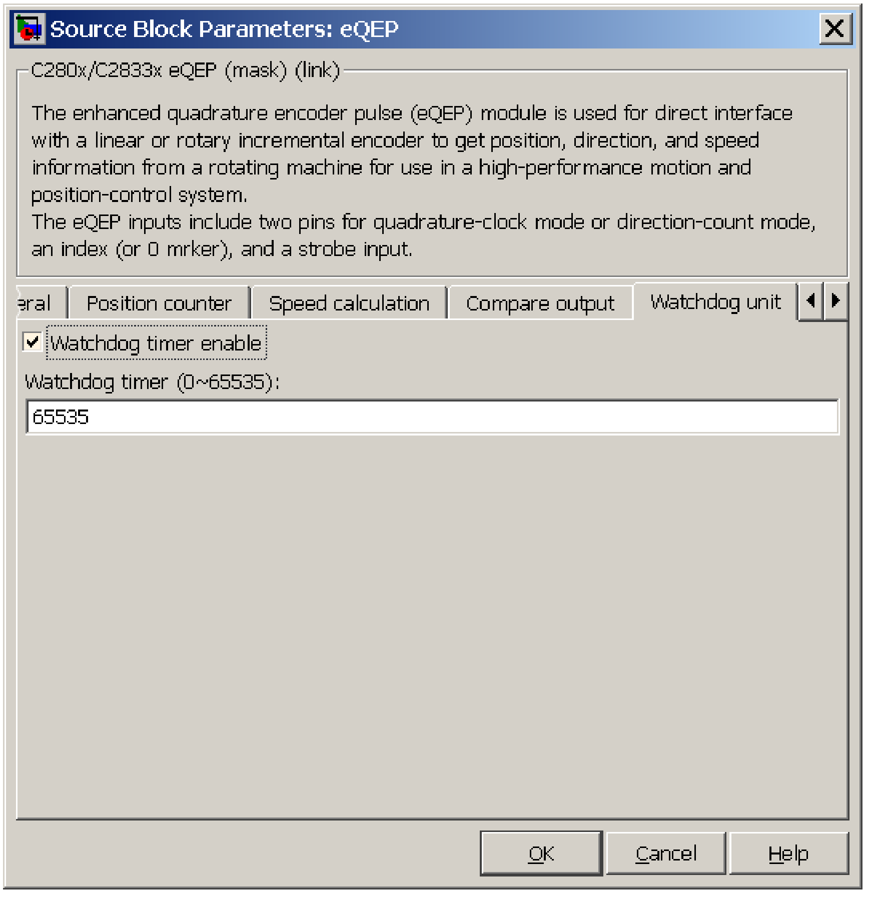
The pulse stretcher logic in the position-compare unit generates a programmable position-compare sync pulse output on the position-compare match.

Enter a value from `1` to `4096` to determine the pulse width of the position-compare sync output signal. The value defaults to `1`.

Polarity of sync output

Choose a value to determine the polarity of the sync output signal: `Active high` (the default) or `Active low`.

Watchdog Unit Pane



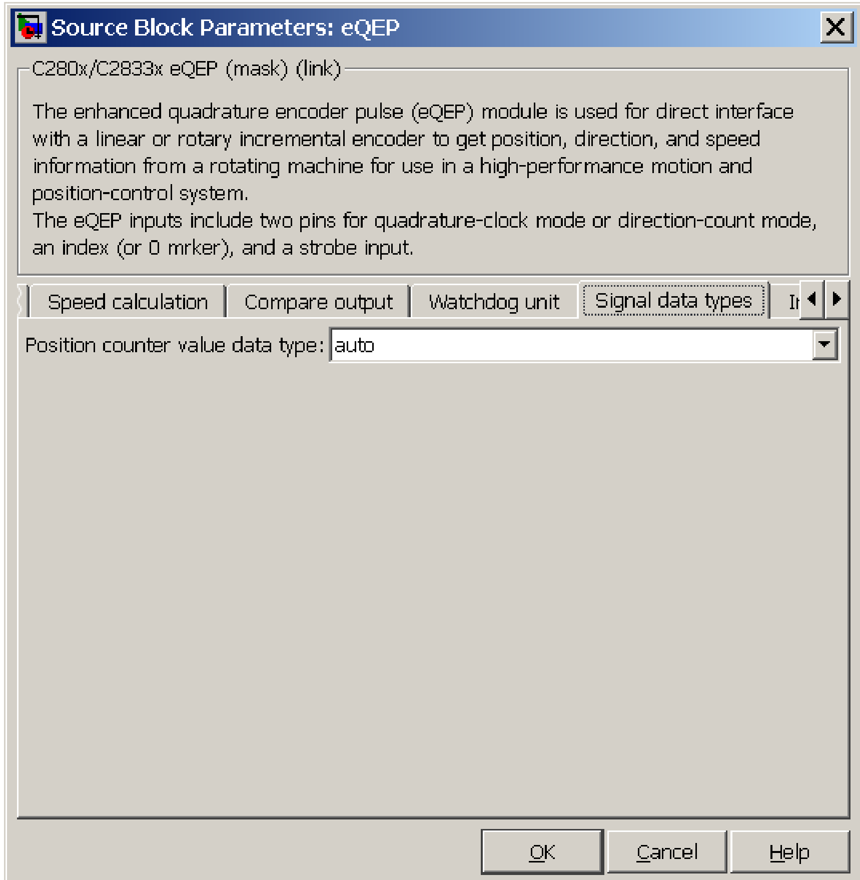
Enable watchdog time out flag via output port

The eQEP peripheral contains a watchdog timer that monitors the quadrature-clock to indicate proper operation of the motion-control system. Select this check box to enable the watchdog time out flag.

Watchdog timer

Enter the time-out value for the watchdog timer. Enter a value from `0` to `65535` (the default).

Signal Data Types Pane



The image above shows the default condition of the **Signal data types** pane. Choosing any of a number of options in other panes of the eQEP dialog box causes a corresponding popup to appear in the **Signal data types** pane.

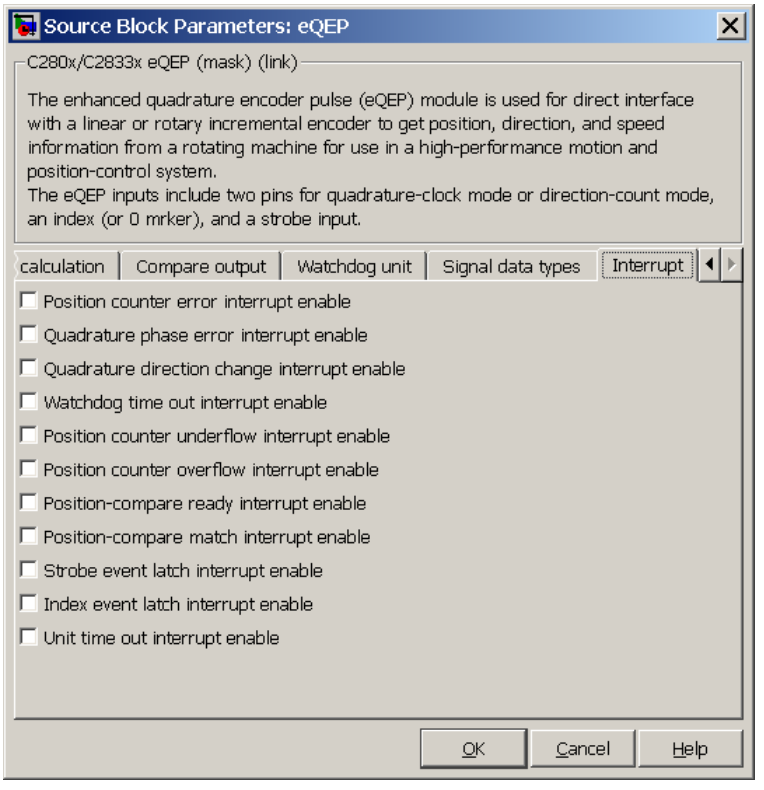
The following table summarizes the options for which you can set the data type in the **Signal data types** pane:

Pane	Option
General	Quadrature direction flag output port
Position counter	Output position counter (selected by default)
	Output position counter error flag
	Output latch position counter on index event
	Output latch position counter on strobe event
Speed calculation	Output capture timer
	Output capture period timer
	Enable and output overflow error flag
	Enable and output direction change error flag
	Output capture timer latched value
	Output capture timer period latched value
	Output position counter latched value
Watchdog unit	Enable watchdog time out flag via output port

The fields that appear on the **Signal data types** pane are named similarly to these options. For example, **Position counter value data type** on the **Signal data types** pane corresponds to the **Output position counter** option on the **Position counter** pane.

For all data type fields, valid data types are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `boolean`.

Interrupt Pane



The image above shows the default condition of the **Interrupt** pane. Interrupts corresponding to specific events are enabled or disabled based on the settings in this pane.

Position counter error interrupt enable

Check this box to enable position counter error interrupts. This checkbox is cleared by default.

Quadrature phase error interrupt enable
Check this box to enable quadrature phase error interrupts. This checkbox is cleared by default.

Quadrature direction change interrupt enable
Check this box to enable quadrature direction change interrupts for changes in the counting direction. This checkbox is cleared by default.

Watchdog timeout interrupt enable
The eQEP Peripheral contains a watchdog timer that monitors the quadrature clock. Check this box to enable watchdog timeout interrupts. This checkbox is cleared by default.

Position counter underflow interrupt enable
Check this box to enable position counter underflow interrupts. This checkbox is cleared by default.

Position counter overflow interrupt enable
Check this box to enable position counter overflow interrupts. This checkbox is cleared by default.

Position-compare ready interrupt enable
Check this box to enable position-compare ready interrupts. This checkbox is cleared by default.

Position-compare match interrupt enable
Check this box to enable position-compare match interrupts. This checkbox is cleared by default.

Strobe event latch interrupt enable
Check this box to enable strobe event latch interrupts. This checkbox is cleared by default.

Index event latch interrupt enable
Check this box to enable index event latch interrupts. This checkbox is cleared by default.

Unit timeout interrupt enable
Check this box to enable unit timeout interrupts. This checkbox is cleared by default.

References

For more information on the QEP module, consult the following documents, available at the Texas Instruments Web site:

- *TMS320x280x, 2801x, 2804x Enhanced Quadrature Encoder Pulse (eQEP) Module Reference Guide*, Literature Number SPRU790
- *Using the Enhanced Quadrature Encoder Pulse (eQEP) Module in TMS320x280x, 28xxx as a Dedicated Capture Application Report*, Literature Number SPRAAH1

See Also

"eQEP" on page 5-907

C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input

Purpose

Configure general-purpose input pins

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

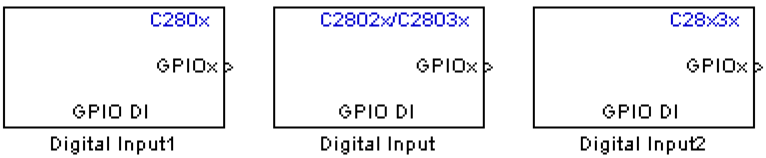
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



This block configures the general-purpose I/O (GPIO) MUX registers that control the operation of GPIO shared pins for digital input. Each I/O port has one MUX register that selects peripheral operation or digital I/O operation (the default). When a pin is configured for digital input, it becomes unavailable for digital output or peripheral operation. You can configure the **Input qualification type** for individual digital input pins. To do so, use the **Peripheral** tab of the Target Preferences block for your processor type.

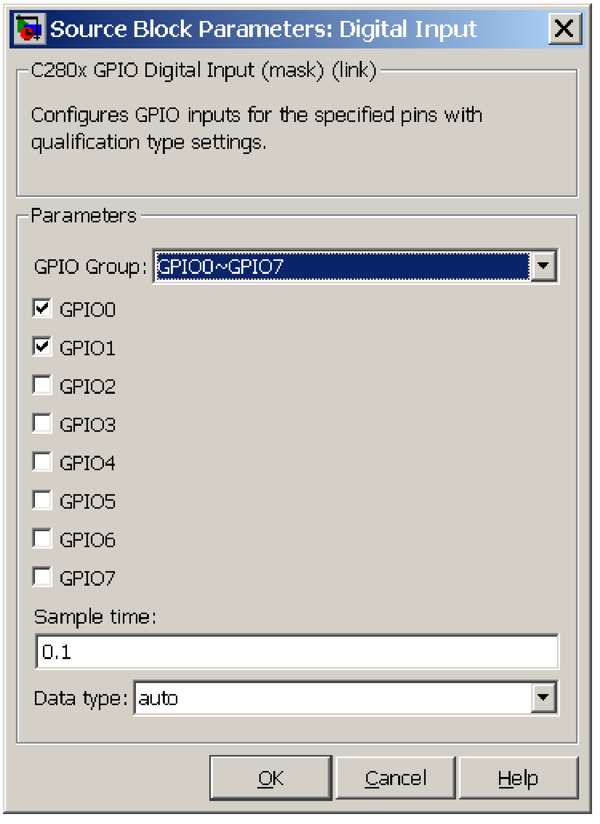
Each processor has a different number of available GPIO pins:

- C280x has 35 GPIO pins
- C2802x has 22 GPIO pins, even though **GPIO group** lists 35
- C2803x has 45 GPIO pins

- C28x3x has 64 GPIO pins

Note To avoid losing any new settings, click **Apply** before changing the **GPIO Group** parameter.

Dialog Box



The dialog boxes for the C2802x and C28x3x processors are similar to that of the C280x, shown in the preceding figure.

GPIO Group
: Select the group of GPIO pins you want to view or configure. For a table of GPIO pins and peripherals, refer to the Texas Instruments documentation for your specific target.

Sample time
: Specify the time interval between output samples. To inherit sample time from the upstream block, set this parameter to `-1`. For more information, refer to the section on "How to Specify the Sample Time" in the Simulink documentation.

Data type
: Specify the data type of the input. The input is read as 16-bit integer, and then cast to the selected data type. Valid data types are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` or `boolean`.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output

"GPIO" on page 5-911

C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output

Purpose

Configure general-purpose input/output pins as digital outputs

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

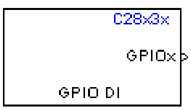
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



Digital Input

Configure individual general-purpose input/output (GPIO) pins to operate as digital outputs. When a pin is configured for digital output, it cannot operate as a digital input or connect to peripheral I/O signals. When you select a pin for digital output, the user interface presents a **Toggle** option that inverts the output signal on the pin.

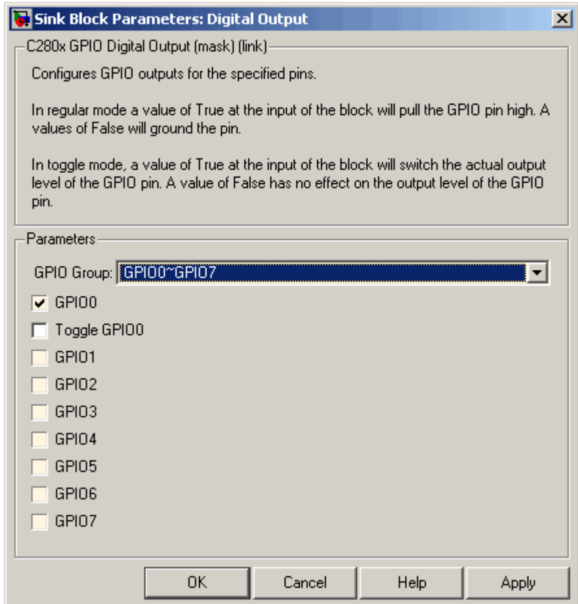
Each processor has a different number of available GPIO pins:

- C280x has 35 GPIO pins
- C2802x has 22 GPIO pins, even though **GPIO group** lists 35
- C2803x has 45 GPIO pins
- C28x3x has 64 GPIO pins

Note To avoid losing any new settings, click **Apply** before changing the **GPIO Group** parameter.

C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output

Dialog Box



The dialog boxes for the C2802x and C28x3x processors are similar to that of the C280x, shown in the preceding figure.

GPIO Group

Select the group of GPIO pins you want to view or configure.

GPIO pins for output

To configure a GPIO pin for digital output, select the checkbox next to it. Refer to the block for a table of all available peripherals for each pin.

A value of `True` at the input of the block drives the selected GPIO pin high. A value of `False` at the input of the block grounds the selected GPIO pin.

Toggle GPIO[bit#]

For each pin selected for output, you can elect to toggle the signal of that pin. In **Toggle** mode, a value of `True` at the input of the block switches the GPIO pin output level. Thus, if the GPIO pin was driven high, in **Toggle** mode, with the value of `True` at the input, the pin output level is driven low. If the GPIO pin was driven low, in **Toggle** mode, with the value of `True` at the input of the block, the same pin output level is driven high. If the input of the block is `False`, there is no effect on the GPIO pin output level.

Note The outputs of this block can be vectorized.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input

"GPIO" on page 5-911

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

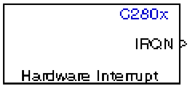
Purpose

Interrupt Service Routine to handle hardware interrupt on C280x/C28x3x processors

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Scheduling

Description



Hardware Interrupt

For many systems, an execution scheduling model based on a timer interrupt is not sufficient for a real-time response to external events. The C280x/C28x3x Hardware Interrupt block addresses this problem by allowing asynchronous processing of interrupts triggered by events managed by other blocks in the C280x/C28x3x DSP Chip Support Library.

The following C280x/C28x3x blocks that can generate an interrupt for asynchronous processing are available in Embedded Coder.

- C280x ADC
- C280x eCAN Receive
- C280x SCI Receive
- C280x SCI Transmit
- C280x SPI Receive
- C280x SPI Transmit

Only one Hardware Interrupt block can be used in a model. To handle multiple interrupts, place a Demux block at the output of the Hardware Interrupt block to direct function calls to the appropriate function-call subsystems.

Vectorized Output

The output of this block is a function call. The size of the function call line equals the number of interrupts the block is set to handle. Each interrupt is represented by four parameters shown on the dialog box of the block. These parameters are a set of four vectors of equal length. Each interrupt is represented by one element from each parameter (four elements total), one from the same position in each of these vectors.

Each interrupt is described by:

- CPU interrupt numbers
- PIE interrupt numbers
- Task priorities
- Preemption flags

So one interrupt is described by a CPU interrupt number, a PIE interrupt number, a task priority, and a preemption flag.

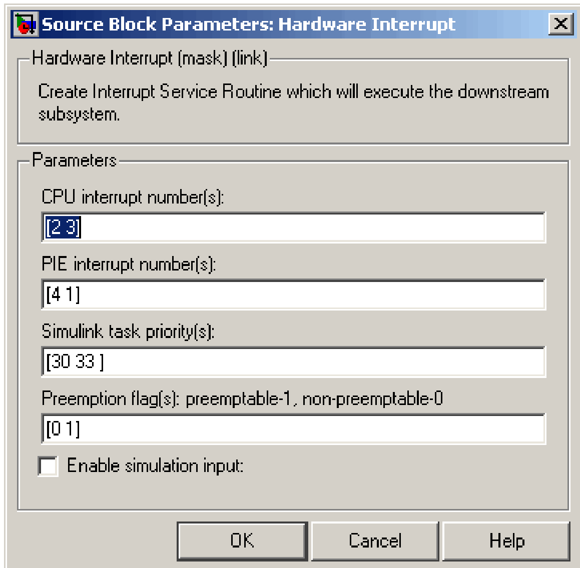
The CPU and PIE interrupt numbers together uniquely specify a single interrupt for a single peripheral or peripheral module. For detailed information about the interrupts, refer to the Texas Instruments documentation for your processor. For example, locate the "PIE MUXed Peripheral Interrupt Vector" or "PIE Peripheral Interrupts" tables in the following Texas Instruments documents:

Processor	Literature Number at ti.com
280x and 28044	SPRU712
C2833x	SPRUFB0, SPRS439
C2802x	SPRUFN3
C2803x	SPRUGL8

The task priority indicates the relative importance tasks associated with the asynchronous interrupts. If an interrupt triggers a higher-priority task while a lower-priority task is running, the execution of the lower-priority task will be suspended while the higher-priority task is executed. The lowest value represents the highest priority. The default priority value of the base rate task is 40, so the priority value for each asynchronously triggered task must be less than 40 for these tasks to suspend the base rate task.

The preemption flag determines whether a given interrupt is preemptable. Preemption overrides prioritization, such that a preemptable task of higher priority can be preempted by a non-preemptable task of lower priority.

Dialog Box



CPU interrupt numbers
Enter a vector of CPU interrupt numbers for the interrupts you want to process asynchronously.

PIE interrupt numbers
Enter a vector of PIE interrupt numbers for the interrupts you want to process asynchronously.

Simulink task priorities
Enter a vector of task priorities for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 5-125 for an explanation of task priorities.

Preemption flags

Enter a vector of preemption flags for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 5-125 for an explanation of preemption flags.

Enable simulation input

Select this check box if you want to be able to test asynchronous interrupt processing in the context of your Simulink software model.

Note Select this check box to enable you to test asynchronous interrupt processing behavior in Simulink software.

References

Detailed information about interrupt processing is in *TMS320x280x DSP System Control and Interrupts Reference Guide*, Literature Number SPRU712B, available at the Texas Instruments Web site.

See Also

The following links refer to block reference pages that require the Embedded Coder software.

C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger,Idle Task

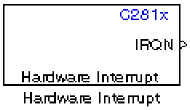
Purpose

Interrupt Service Routine to handle hardware interrupt

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Scheduling

Description



For many systems, an execution scheduling model based on a timer interrupt is not sufficient for a real-time response to external events. The C281x Hardware Interrupt block addresses this problem by allowing for the asynchronous processing of interrupts triggered by events managed by other blocks in the C281x DSP Chip Support Library.

The following C281x blocks that can generate an interrupt for asynchronous processing are available fromEmbedded Coder:

- C281x ADC
- C281x CAP
- C281x eCAN Receive
- C281x Timer
- C281x SCI Receive
- C281x SCI Transmit
- C281x SPI Receive
- C281x SPI Transmit

Only one Hardware Interrupt block can be used in a model. To handle multiple interrupts, place a Demux block at the output of the Hardware Interrupt block to direct function calls to the appropriate function-call subsystems.

Vectorized Output

The output of this block is a function call. The size of the function call line equals the number of interrupts the block is set to handle. Each interrupt is represented by four parameters shown on the dialog box of the block. These parameters are a set of four vectors of equal length.

Each interrupt is represented by one element from each parameter (four elements total), one from the same position in each of these vectors.

Each interrupt is described by:

- CPU interrupt numbers
- PIE interrupt numbers
- Task priorities
- Preemption flags

So one interrupt is described by a CPU interrupt number, a PIE interrupt number, a task priority, and a preemption flag.

The CPU and PIE interrupt numbers together uniquely specify a single interrupt for a single peripheral or peripheral module. The following table maps CPU and PIE interrupt numbers to these peripheral interrupts.

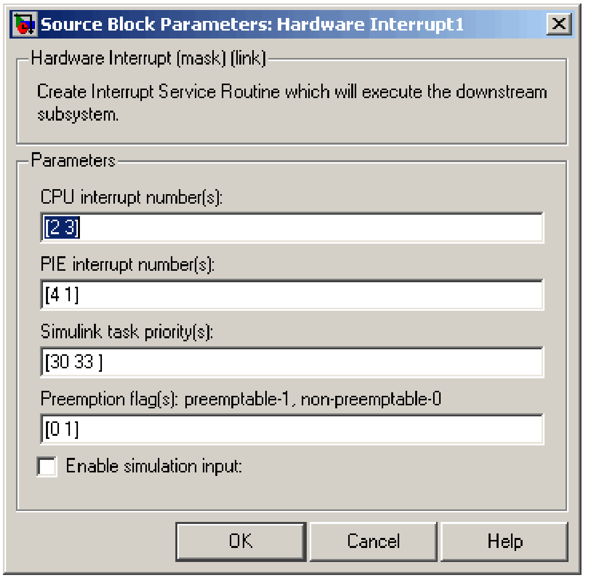
C281x Peripheral Interrupt Vector Values

	Row numbers = CPU values / Column numbers = PIE values							
	8	**7**	**6**	**5**	**4**	**3**	**2**	**1**
1	WAKEINT (LPM/WD)	TINT0 (TIMER 0)	ADCINT (ADC)	XINT2	XINT1	Reserved	PDPINTB (EV-B)	PDPINTA (EV-A)
2	Reserved	T1OFINT (EV-A)	T1UFINT (EV-A)	T1CINT (EV-A)	T1PINT (EV-A)	CMP3INT (EV-A)	CMP2INT (EV-A)	CMP1INT (EV-A)
3	Reserved	CAPINT3 (EV-A)	CAPINT2 (EV-A)	CAPINT1 (EV-A)	T2OFINT (EV-A)	T2UFINT (EV-A)	T2CINT (EV-A)	T2PINT (EV-A)
4	Reserved	T3OFINT (EV-B)	T3UFINT (EV-B)	T3CINT (EV-B)	T3PINT (EV-B)	CMP6INT (EV-B)	CMP5INT (EV-B)	CMP4INT (EV-B)
5	Reserved	CAPINT6 (EV-B)	CAPINT5 (EV-B)	CAPINT4 (EV-B)	T4OFINT (EV-B)	T4UFINT (EV-B)	T4CINT (EV-B)	T4PINT (EV-B)
6	Reserved	Reserved	MXINT (McBSP)	MRINT (McBSP)	Reserved	Reserved	SPITXINTA (SPI)	SPIRXINTA (SPI)
7	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
8	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
9	Reserved	Reserved	ECAN1INT (CAN)	ECAN0INT (CAN)	SCITXINTB (SCI-B)	SCIRXINTB (SCI-B)	SCITXINTA (SCI-A)	SCIRXINTA (SCI-A)
10	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
11	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
12	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

The task priority indicates the relative importance tasks associated with the asynchronous interrupts. If an interrupt triggers a higher-priority task while a lower-priority task is running, the execution of the lower-priority task will be suspended while the higher-priority task is executed. The lowest value represents the highest priority. Note that the default priority value of the base rate task is 40, so the priority value for each asynchronously triggered task must be less than 40 for these tasks to actually cause the suspension of the base rate task.

The preemption flag determines whether a given interrupt is preemptable or not. Preemption overrides prioritization, such that a preemptable task of higher priority can be preempted by a non-preemptable task of lower priority.

Dialog Box



CPU interrupt numbers

Enter a vector of CPU interrupt numbers for the interrupts you want to process asynchronously.

See the table of C281x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

PIE interrupt numbers

Enter a vector of PIE interrupt numbers for the interrupts you want to process asynchronously.

See the table of C281x Peripheral Interrupt Vector Values for a mapping of CPU interrupt number to interrupt names.

Simulink task priorities

Enter a vector of task priorities for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 5-129 for an explanation of task priorities.

Preemption flags

Enter a vector of preemption flags for the interrupts you want to process asynchronously.

See the discussion of this block's "Vectorized Output" on page 5-129 for an explanation of preemption flags.

Enable simulation input

Select this check box if you want to be able to test asynchronous interrupt processing in the context of your Simulink software model.

Note Use this check box to enable you to test asynchronous interrupt processing behavior in Simulink software.

References

Detailed information interrupt processing is in *TMS320x281x DSP System Control and Interrupts Reference Guide*, Literature Number SPRU078C, available at the Texas Instruments Web site.

See Also

The following links to block reference pages require that Embedded Coder is installed.

C281x Software Interrupt Trigger,C281x Timer, Idle Task

Purpose

Configure inter-integrated circuit (I2C) module to receive data from I2C bus

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

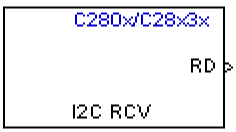
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

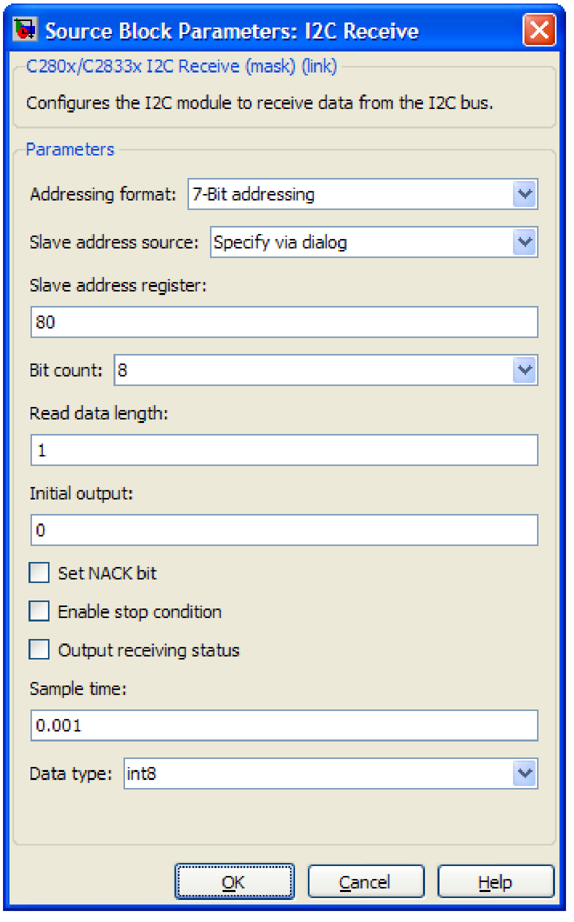
Description



I2C Receive

Configure the I2C module to receive data from the two-wire I2C serial bus.

Dialog Box



Addressing format

The I2C receive block supports the **7–Bit addressing**, **10–Bit addressing**, and **Free data format**. The default setting is **7–Bit addressing**.

Slave address source

Select the method for setting the slave address register of the I2C slave. Selecting **Specify via dialog** displays **Slave address**

register parameter. Selecting **Input port** enables definition of the address register via the input port. The default setting is **Specify via dialog**.

Slave address register
: When you select **Specify via dialog**, enter a value for the **Slave address register**. The default value is **80**. This field takes a decimal value.

Bit Count
: Set the bit count to 1 through 8. The default setting is **8**.

Read data length
: Set the length of the read data. The default value is **1**.

Initial output
: Set the value the I2C node outputs to the model before it has received any data.

The default value is `0`.

NACK bit generation
: Select this parameter to generate a no-acknowledge bit (NACK) during the I2C acknowledge cycle and ignore new bits from the transmitting I2C node. The default setting is disabled (not selected).

Enable stop condition
: Enable the I2C Receive Block in master mode to send a STOP message to the I2C Transmit block while it is in slave mode. The default setting is disabled (not selected).

Output receiving status
: Selecting this parameter creates a status output that indicates when the I2C receive block is receiving a message. The default setting is disabled (not selected).

Sample time
: Set the sample time for the block's input sampling. To execute this block asynchronously, set **Sample Time** to `-1`, and refer to "Asynchronous Interrupt Processing" for a discussion of block

placement and other necessary settings. The default value is **0.001**.

Data type

Type of data in the data vector. The length of the vector for the received message is at most 8 bytes. If the message is less than 8 bytes, the data buffer bytes are right-aligned in the output. You can set this parameter to int8, uint8, int16, uint16, int32, or uint32. The default setting is **int8**.

References

For detailed information on the I2C module, see:

- The *TMS320x28xx, 28xxx Inter-Integrated Circuit (I2C) Module Reference Guide*, Literature Number SPRU721, available at the Texas Instruments Web site, www.ti.com.
- The *Philips Semiconductors Inter-IC bus (I2C-bus) specification version 2.1* is available on the Philips Semiconductors Web site at http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf.

See Also

C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit

"I2C" on page 5-893

Purpose

Configure inter-integrated circuit (I2C) module to transmit data to I2C bus

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

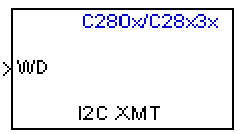
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description

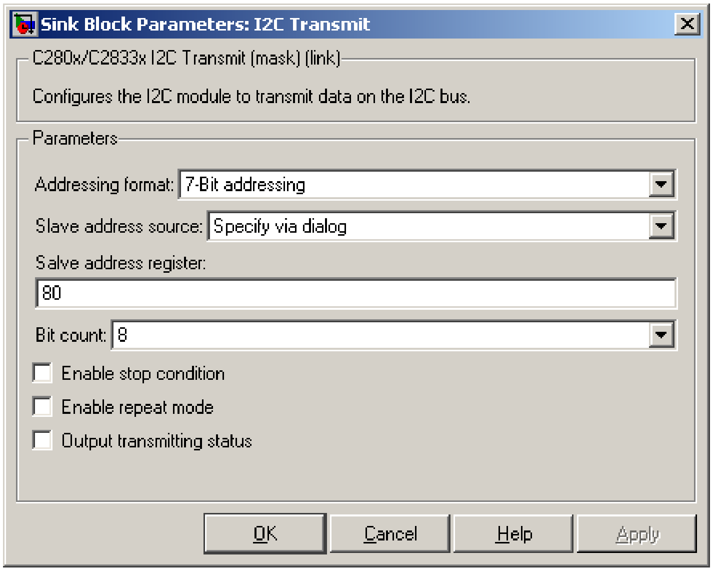


I2C Transmit

Configure the I2C module to transmit data to the two-wire I2C serial bus.

Note You can use this block to configure the I2C settings under the Peripherals tab of the target preference blocks for the F2808 eZdsp, and F28335 eZdsp boards.

Dialog Box



Addressing format

The I2C transmit block supports the **7–Bit addressing**, **10–Bit addressing**, and **Free data format**. The default setting is **7–Bit addressing**.

Slave address source

Select the method for setting the slave address register of the I2C slave. Selecting **Specify via dialog** displays **Slave address register** parameter . Selecting **Input port** enables definition of the address register via the input port. The default setting is **Specify via dialog**.

Slave address register

When you select **Specify via dialog**, enter a value for the **Slave address register**. The default value is **80**.

Bit Count

Set the bit count to 1 through 8. The default setting is **8**.

Enable stop condition

Selecting this parameter enables the transmitter to accept a STOP condition from the C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive block. The default setting is disabled (not selected).

Enable repeat mode

When you enable repeat mode, the I2C module retransmits the same data until it detects a stop or start condition. If you use this mode, also consider selecting **Enable stop condition**.

If you disable repeat mode, the I2C module operates in standard mode, sending a specific number of data values once.

The default setting is disabled (not selected).

Output transmitting status

Selecting this parameter creates a status output that indicates when the I2C transmit block is transmitting a message. The default setting is disabled (not selected).

References

For detailed information on the I2C module, see:

- The *TMS320x28xx, 28xxx Inter-Integrated Circuit (I2C) Module Reference Guide*, Literature Number SPRU721, available at the Texas Instruments Web site, www.ti.com.
- The *Philips Semiconductors Inter-IC bus (I2C-bus) specification version 2.1* is available on the Philips Semiconductors Web site at http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf.

See Also

C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive

"I2C" on page 5-893

C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive

Purpose

Receive data on target via serial communications interface (SCI) from host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

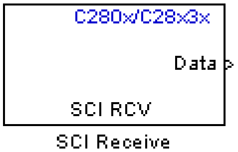
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



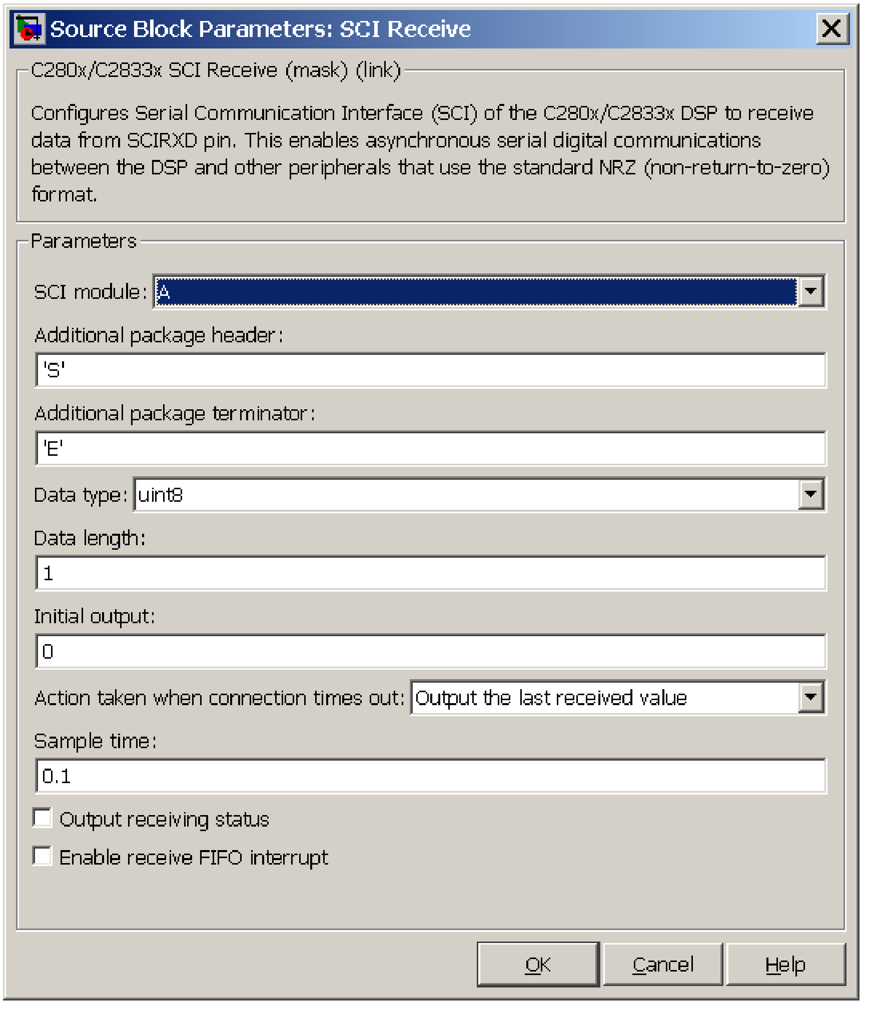
SCI Receive

The SCI Receive block supports asynchronous serial digital communications between the target and other asynchronous peripherals in nonreturn-to-zero (NRZ) format. This block configures the DSP target to receive scalar or vector data from the COM port via the target's COM port.

Note For any given model, you can have only one SCI Receive block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

Many SCI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Dialog Box



SCI module

SCI module to be used for communications.

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 may use any 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. To specify a null value (no package header), enter two single quotes alone.

Note Any additional packager header or terminator must match the additional package header or terminator specified in the host 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 may use any 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. To specify a null value (no package terminator), enter two single quotes alone.

Data type

Data type of the output data. Available options are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

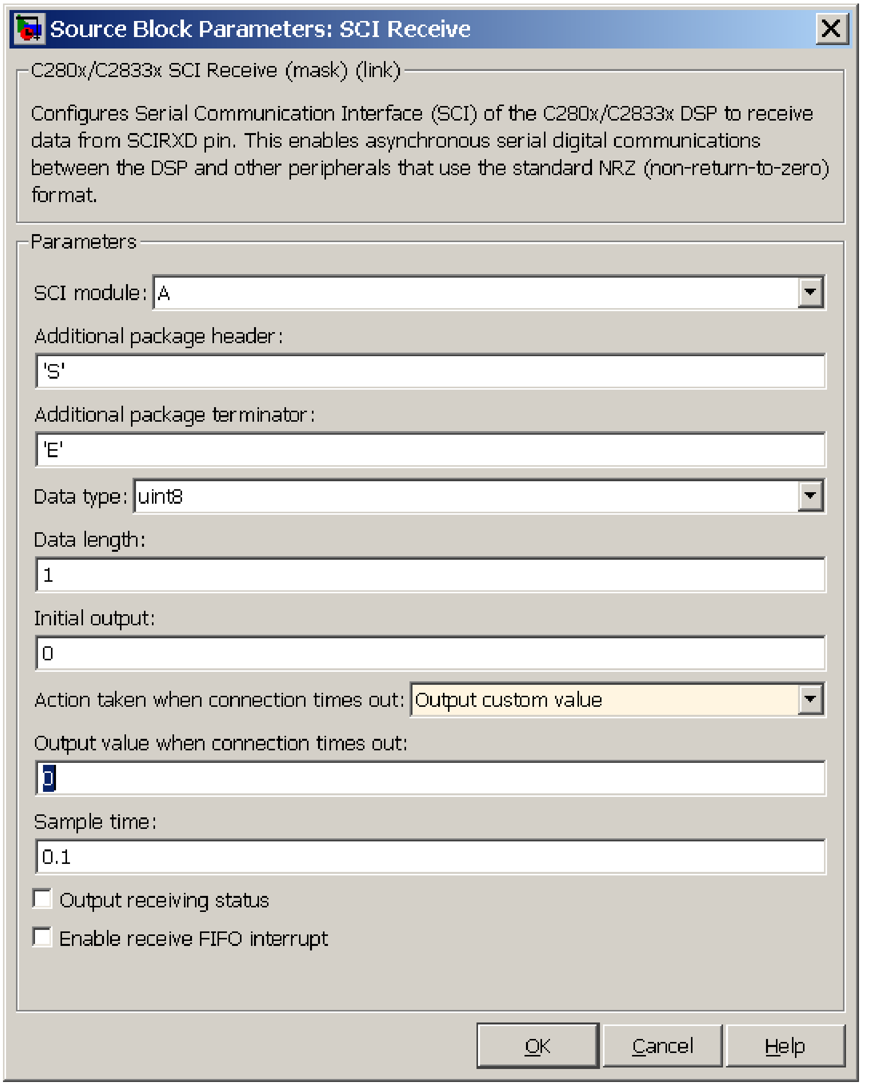
Data length

How many of **Data type** the block will receive (not bytes). Anything more than 1 is a vector. The data length is inherited from the input (the data length originally input to the host-side 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 timeout
Specify what to output if a connection time-out occurs. If “Output the last received value” is selected, the last received value is what is output, unless none has been received yet, in which case the **Initial output** is considered the last received value.

If you select "Output custom value", use the "Output value when connection times out" field to set the custom value.

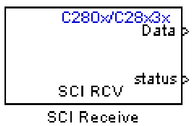

Source Block Parameters: SCI Receive
C280x/C2833x SCI Receive (mask) (link)
Configures Serial Communication Interface (SCI) of the C280x/C2833x DSP to receive data from SCIRXD pin. This enables asynchronous serial digital communications between the DSP and other peripherals that use the standard NRZ (non-return-to-zero) format.
Parameters
SCI module: A
Additional package header:
'S'
Additional package terminator:
'E'
Data type: uint8
Data length:
1
Initial output:
0
Action taken when connection times out: Output custom value
Output value when connection times out:
0
Sample time:
0.1
Output receiving status
Enable receive FIFO interrupt
OK
Cancel
Help

Sample time

Sample time, T_s, for the block's input sampling. To execute this block asynchronously, set **Sample Time** to -1, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

Output receiving status

When this field is checked, the SCI Receive block adds another output port for the transaction status, and appears as shown in the following figure.



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

Enable receive FIFO interrupt

If this option is selected, an interrupt is posted when FIFO is full, allowing the subsystem to take some sort of action (for example, read data as soon as it is received). If this option is cleared, the block stays in polling mode. If the block is in polling mode and not blocking, it checks the FIFO to see if there is data to read. If data is present, it reads and outputs. If no data is present, it continues. If the block is in polling mode and blocking, it waits until data is available to read (after data length is reached).

Receive FIFO interrupt level

This parameter is enabled when the **Enable receive FIFO interrupt** option is selected. Select an interrupt level from 0 to 16. The default level is 0.

References

For detailed information on the SCI module, see *TMS320x281x, 280x DSP Serial Communication Interface (SCI) Reference Guide*, Literature Number SPRU051B, available at the Texas Instruments Web site.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit, C280x/C2802x/C2803x/C28x3x Hardware Interrupt

"SCI_A, SCI_B, SCI_C" on page 5-900

C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit

Purpose

Transmit data from target via serial communications interface (SCI) to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

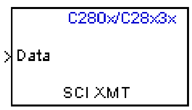
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



SCI Transmit

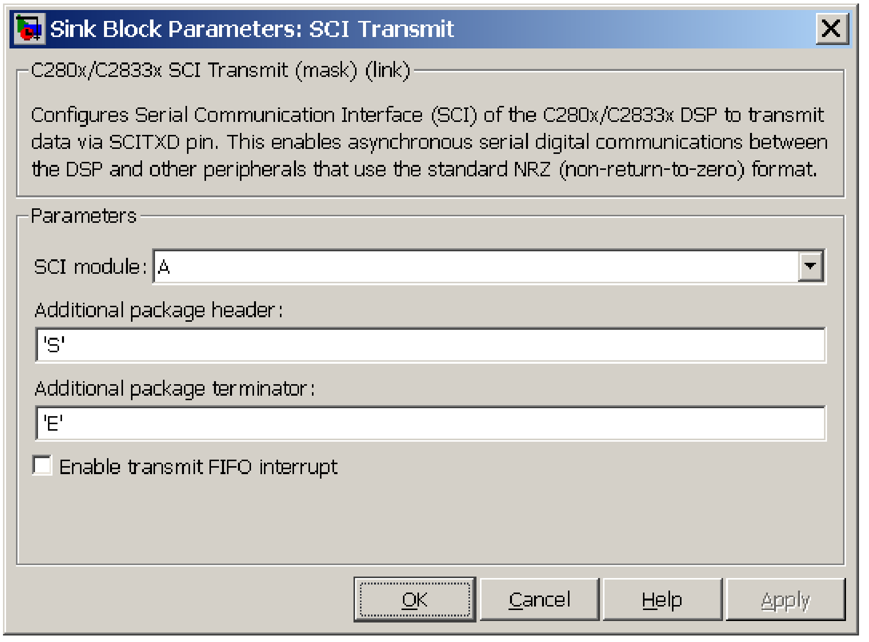
The SCI Transmit block transmits scalar or vector data in `int8` or `uint8` format from the target's COM ports in nonreturn-to-zero (NRZ) format. You can specify how many of the six target COM ports to use. The sampling rate and data type are inherited from the input port. The data type of the input port must be one of the following: single, int8, uint8, int16, uint16, int32, uint32. If no data type is specified, the default data type is `uint8`.

Note For any given model, you can have only one SCI Transmit block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

Many SCI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Fixed-point inputs are not supported for this block.

Dialog Box



SCI module

SCI module to be used for communications.

Additional package header

This field specifies the data located at the front of the sent 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 may use any 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. To specify a null value (no package header), enter two single quotes alone.

Note Any additional packager header or terminator must match the additional package header or terminator specified in the host SCI Receive block.

Additional package terminator

This field specifies the data located at the end of the sent data package, which is not part of the data being transmitted, and generally indicates end of data. The additional package terminator must be an ASCII value. You may use any 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. To specify a null value (no package terminator), enter two single quotes alone.

Enable transmit FIFO interrupt

If checked, an interrupt is posted when FIFO is full, allowing the subsystem to take some sort of action.

References

For detailed information on the SCI module, see *TMS320x281x, 280x DSP Serial Communication Interface (SCI) Reference Guide*, Literature Number SPRU051B, available at the Texas Instruments Web site.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

"SCI_A, SCI_B, SCI_C" on page 5-900

C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger

Purpose

Generate software triggered nonmaskable interrupt

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

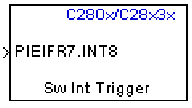
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



Software Interrupt Trigger

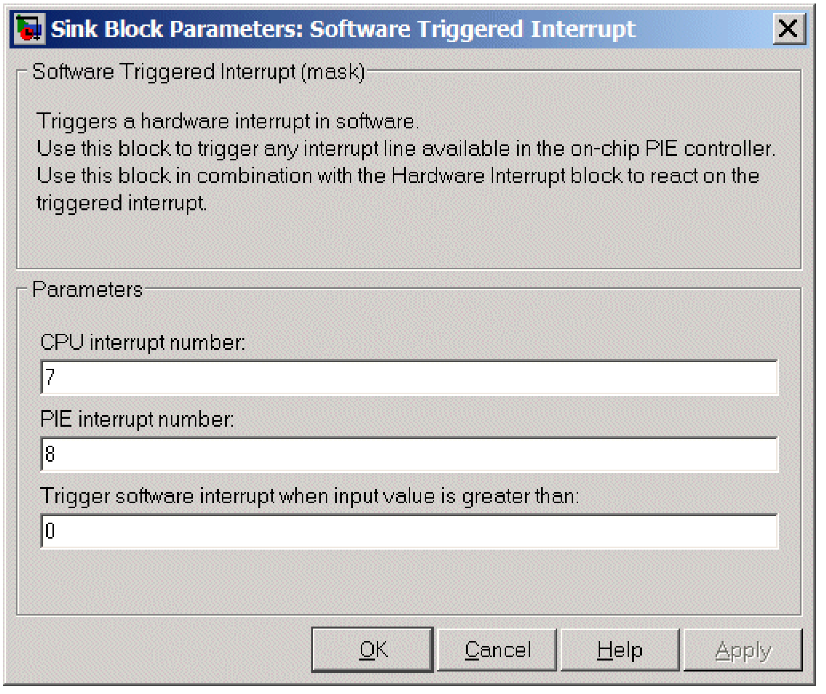
When you add this block to a model, the block polls the input port for the input value. When the input value is greater than the value in **Trigger software interrupt when input value is greater than**, the block posts the interrupt to a Hardware Interrupt block in the model.

To use this block, add a Hardware Interrupt block to your model to process the software triggered interrupt from this block into an interrupt service routine on the processor. Set the interrupt number in the Hardware Interrupt block to the value you set here in **CPU interrupt number**.

The CPU and PIE interrupt numbers together specify a single interrupt for a single peripheral or peripheral module. The following table maps CPU and PIE interrupt numbers to these peripheral interrupts. The row numbers are CPU values and the column numbers are the PIE values.

Note Fixed-point inputs are not supported for this block.

Dialog Box



CPU interrupt number

Specify the interrupt to which the block responds. Interrupt numbers are integers ranging from 1 to 12.

PIE interrupt number

Enter an integer value from 1 to 8 to set the Peripheral Interrupt Expansion (PIE) interrupt number.

Trigger software interrupt when input value is greater than:

Sets the value above which the block posts an interrupt. Enter the value for the level that indicates that the interrupt is asserted by a requesting routine.

C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger

References

For detailed information about interrupt processing, see *TMS320x280x DSP System Control and Interrupts Reference Guide*, SPRU712B, available at the Texas Instruments Web site.

See Also

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive

Purpose

Receive data via serial peripheral interface (SPI) on target

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

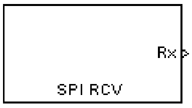
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



SPI Receive

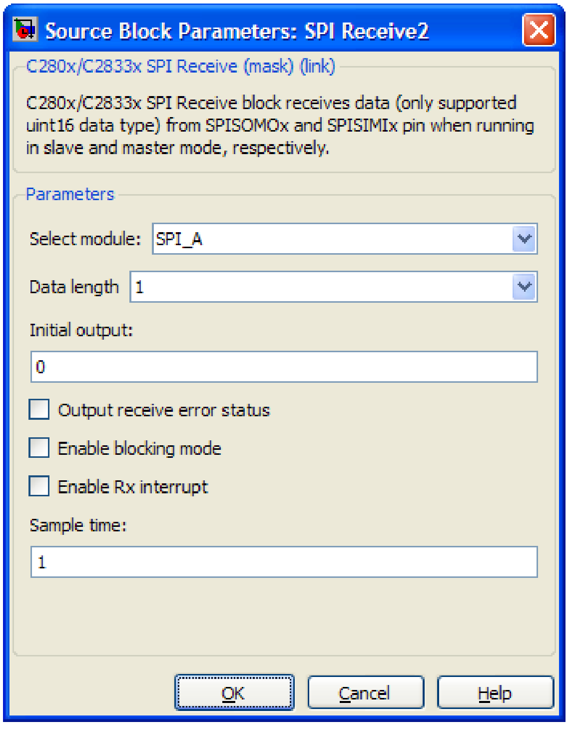
The SPI Receive block supports synchronous, serial peripheral input/output port communications between the DSP controller and external peripherals or other controllers. The block can run in either slave or master mode.

In master mode, the SPISIMO pin transmits data and the SPISOMI pin receives data. When master mode is selected, the SPI initiates the data transfer by sending a serial clock signal (SPICLK), which is used for the entire serial communications link. Data transfers are synchronized to this SPICLK, which enables both master and slave to send and receive data simultaneously. The maximum for the clock is one quarter of the DSP controller's clock frequency.

For any given model, you can have only one SPI Receive block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

Note Many SPI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Dialog Box



Select module

Select the SPI module to be used for communications. Each processor has a different number of modules.

Data length

Specify how many `uint16`s are expected to be received. Select 1 through 16.

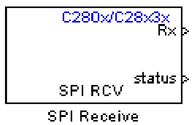
Initial output

Set the value the SPI node outputs to the model before it has received any data.

The default value is `0`.

Enable blocking mode
If this option is selected, system waits until data is received before continuing processing.

Output receive error status
When this field is checked, the SPI Receive block adds another output port for the transaction status, and appears as shown in the following figure.



Error status may be one of the following values:

- 0: No errors
- 1: Data loss occurred, (Overrun: when FIFO disabled, Overflow when FIFO enabled)
- 2: Data not ready, a time out occurred while the block was waiting to receive data

Post interrupt when data is received
Check this check box to post an asynchronous interrupt when data is received.

Sample time
Sample time, T_s, for the block's input sampling. To execute this block asynchronously, set **Sample Time** to -1, check the **Post interrupt when message is received** box, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

"SPI_A, SPI_B, SPI_C, SPI_D" on page 5-904

C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit

Purpose

Transmit data via serial peripheral interface (SPI) to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

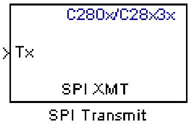
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



SPI Transmit

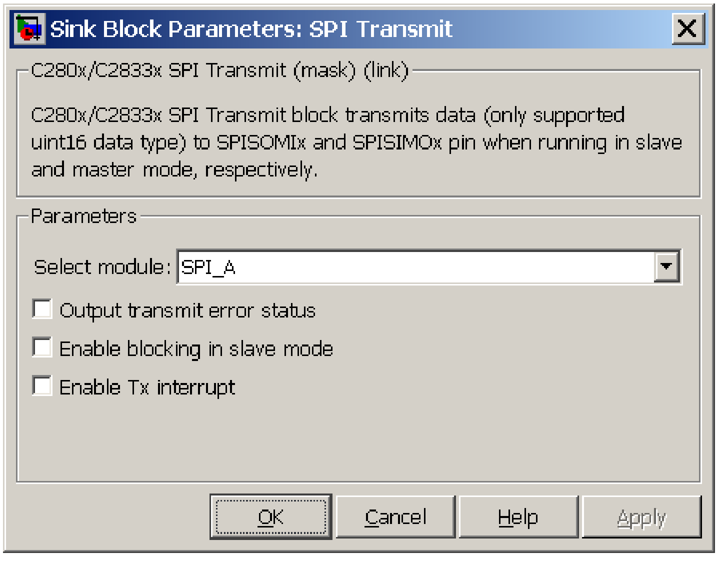
The SPI Transmit supports synchronous, serial peripheral input/output port communications between the DSP controller and external peripherals or other controllers. The block can run in either slave or master mode. In master mode, the SPISIMO pin transmits data and the SPISOMI pin receives data. When master mode is selected, the SPI initiates the data transfer by sending a serial clock signal (SPICLK), which is used for the entire serial communications link. Data transfers are synchronized to this SPICLK, which enables both master and slave to send and receive data simultaneously. The maximum for the clock is one quarter of the DSP controller's clock frequency.

The sampling rate is inherited from the input port. The supported data type is `uint16`.

Note For any given model, you can have only one SPI Transmit block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

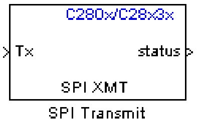
Many SPI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Dialog Box



Select module
Select the SPI module to be used for communications. Each processor has a different number of modules.

Output transmit error status
When this field is checked, the SPI Transmit block adds another output port for the transaction status, and appears as shown in the following figure.



Error status may be one of the following values:

- 0: No errors
- 1: A time-out occurred while the block was transmitting data
- 2: There is an error in the transmitted data (for example, header or terminator don't match, length of data expected is too big or too small)

Enable blocking mode
If this option is selected, system waits until data is sent before continuing processing.

Post interrupt when data is transmitted
Check this check box to post an asynchronous interrupt when data is transmitted.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

"SPI_A, SPI_B, SPI_C, SPI_D" on page 5-904

Purpose

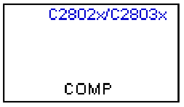
Compare two input voltages on comparator pins

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

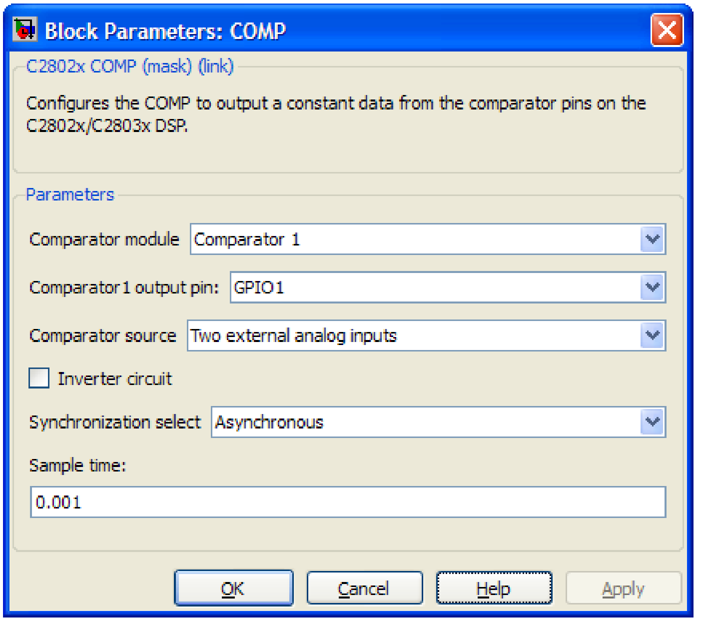
Description



COMP

Configures the COMP to output a constant data from the comparator pins on the DSP.

Dialog Box



Comparator module
Select which comparator module the block configures. Use only one block per module.

Comparator # output pin
Select the GPIO pin to use for the comparator output.

Comparator source
Select `Two external analog inputs` to compare the voltage of **Input Pin A** with **Input Pin B**. Select `One external analog inputs` to compare the voltage of **Input Pin A** with the output of a DAC reference located in the comparator. For more information, see the "DAC Reference" section of the *TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator*.

The comparator source outputs 1 if **Input Pin A** has a value greater than **Input Pin B** or the 10-bit DAC reference. Otherwise it outputs 0.

Inverter circuit
Apply a logical NOT to the output of the comparator source. For example, when the comparator source outputs 1, the inverter circuit changes it to 0.

Synchronization select
Select `Asynchronous` to pass the asynchronous version of the comparator output. Select `Synchronous` to pass the synchronous version of the comparator output. Selecting `Synchronous` enables the **Qualification period** option.

Qualification period
Qualify changes in the comparator output before passing them along. The `Passed through` setting passes changes in the comparator value along without qualifying them. The `consecutive clocks` settings pass changes in the comparator value along after receiving the specified number of consecutive samples with the same value. Use this setting to prevent intermittent and spurious changes in the comparator output.

Sample time

Specify the time interval between samples. To inherit sample time from the upstream block, set this parameter to -1.

References

TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator, Literature Number: SPRUGE5, from the Texas Instruments Web site.

Purpose

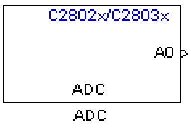
Configure ADC to sample analog pins and output digital data

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

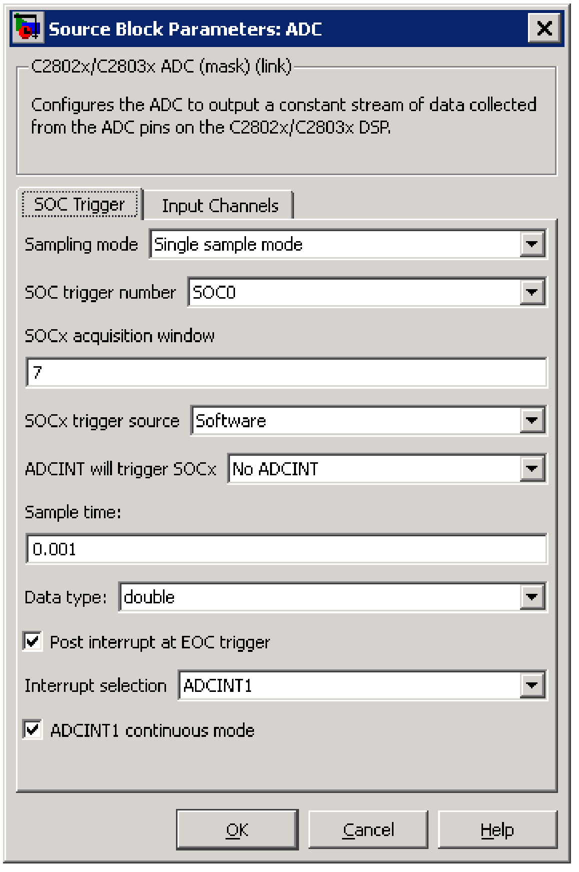
Description



Configures the ADC to output a constant stream of data collected from the ADC pins on the DSP.

An ADC block allows for reading one ADC channel. Use multiple ADC blocks to read multiple ADC channels.

Dialog Box



Sampling mode

Select **Single sample mode** to sample two signals sequentially. Select **Simultaneous sample mode** to sample the two signals with a minimal delay between the samples.

SOC trigger number

Identify the start-of-conversion trigger by number. In single sampling mode, you can select an individual trigger. In simultaneous sampling mode, you can select triggers in pairs.

SOCx acquisition window

Define the length of the acquisition period, the acquisition window, in sample cycles. The minimal value for this parameter is 7 cycles. For more information, see the "ADC Acquisition (Sample and Hold) Window" section of the *TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator Reference Guide*.

SOCx trigger source

Select the source that triggers the start of conversion. The following types of inputs are available:

- Software
- CPU Timers 0/1/2 interrupts
- XINT2 SOC
- ePWM1-7 SOCA and SOCB

If you set **SOCx trigger source** to `XINT2_XINT2SOC`, use the **XINT2SOC external pin** parameter in the Target Preferences block to define the external GPIO pin that triggers the start of conversion. **XINT2SOC external pin** is located under the Target Preferences block's Peripherals tab, on the ADC pane.

ADCINT will trigger SOCx

At the end of conversion, use the `ADCINT1` or `ADCINT2` interrupt to trigger a start of conversion (SOC). This loop creates a continuous sequence of conversions. The default selection, `No ADCINT` disables this parameter.

Sample time

Specify the time interval between samples. To inherit sample time from the upstream block, set this parameter to -1.

Data type

Select the data type of the digital output data. You can choose from the options `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`.

Post interrupt at EOC trigger

Post interrupts when the ADC triggers EOC pulses. When you select this option, the dialog box displays the **Interrupt selection** and **ADCINT# continuous mode** options. For more information, see the "EOC and Interrupt Operation" section of the *TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator Reference Guide*.

Interrupt selection

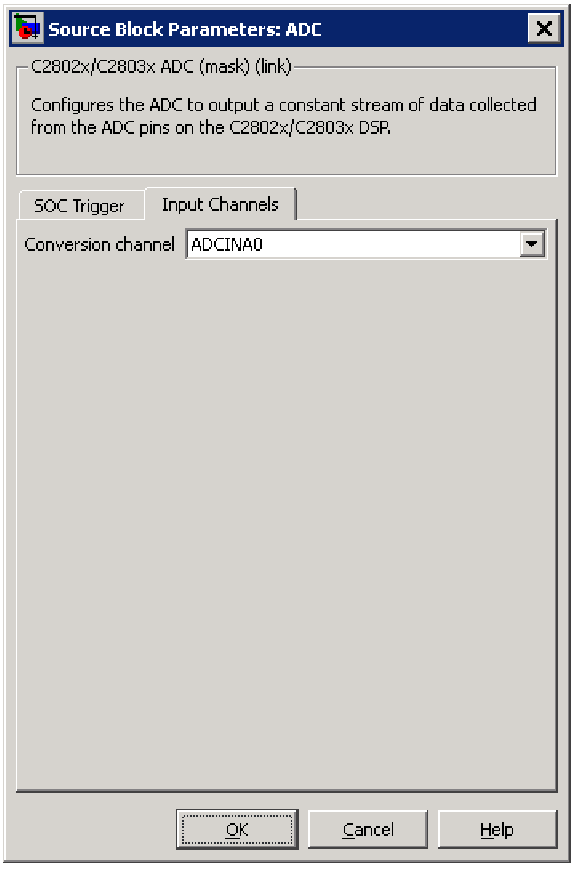
Select which interrupt the ADC posts after triggering an EOC pulse.

ADCINT1 continuous mode
ADCINT2 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.

Input Channels — Conversion channel

Select the input channel to which this ADC conversion applies.



References

TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator, Literature Number: SPRUGE5, from the Texas Instruments Web site.

See Also

C280x/C2802x/C2803x/C28x3x/c2834x ePWM

C280x/C2802x/C2803x/C28x3x Hardware Interrupt

“Configuring Acquisition Window Width for ADC Blocks”

"ADC" on page 5-883

C2802x/C2803x AnalogIO Input

Purpose

Configure pin, sample time, and data type for analog input

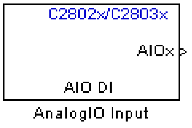
Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

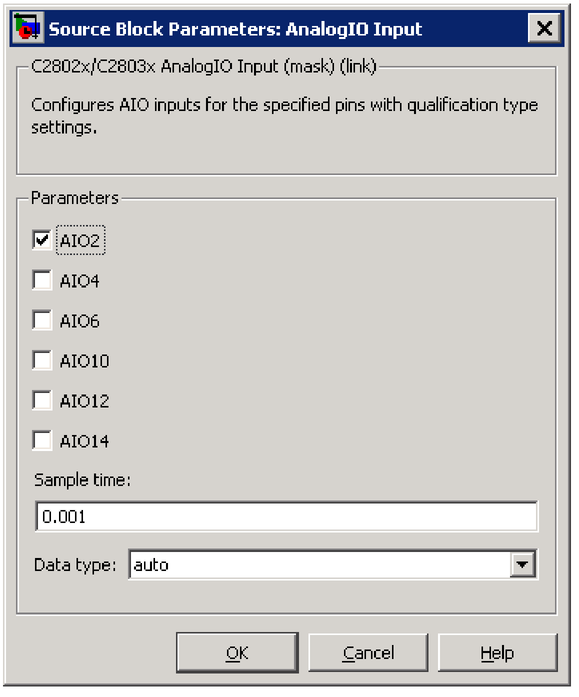
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Description

Use this block to sample the Analog IO input pins on the C2802x processor for a positive voltage and output the results.



Dialog Box



Parameters (Input pins)
Select the input pins to sample.

Sample time
Specify the time interval between samples. To inherit sample time from the upstream block, set this parameter to -1.

Data type
Select the data type of the digital output data. If you select `auto`, the block automatically selects the correct data type for your model. You can also manually select a data type. You can choose from the options `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`.

See Also

C2802x/C2803x AnalogIO Output

C2802x/C2803x AnalogIO Output

Purpose

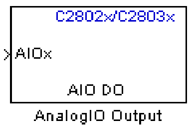
Configure Analog IO to output analog signals on specific pins

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

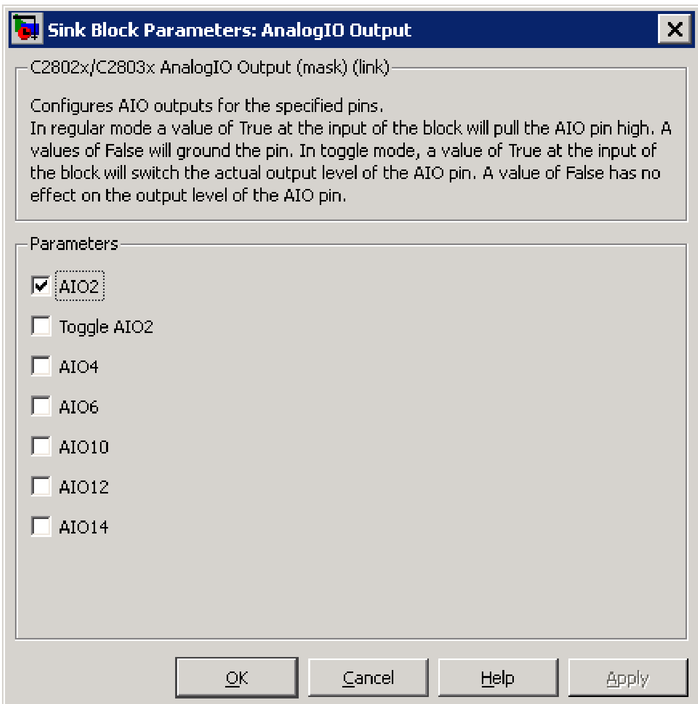
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Description



Configures the Analog IO output pins for the specified pins. In regular mode, a value of True at the input of the block pulls the Analog IO pin high. A value of False grounds the pin. In toggle mode, a value of True at the input of the block switches the actual output level of the Analog IO pin. A value of False does not affect on the output level of the Analog IO pin.

Dialog Box



Parameters (Output Pins)
Select the analog output pins that express the value of the digital input on **AIOx**. Selecting **Toggle** inverts the output voltage levels of the pins.

See Also

C2802x/C2803x AnalogIO Input

C2803x LIN Receive

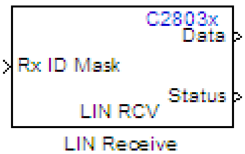
Purpose

Receive data via local interconnect network (LIN) module on target

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Description



The Local Interconnect Network (LIN) bus implements a serial communications protocol for distributed automotive and industrial applications. In particular, LIN serves low cost applications that do not require the bandwidth or robustness provided by the CAN protocol. For more information about LIN, see http://www.lin-subbus.org/.

The LIN Receive block configures the target to receive scalar or vector data from the LINRX or LINTX pins.

Each C2803x target has one LIN module. Your model can only contain one LIN Transmit and one LIN Receive block per module.

The C2803x LIN Transmit block takes three inputs:

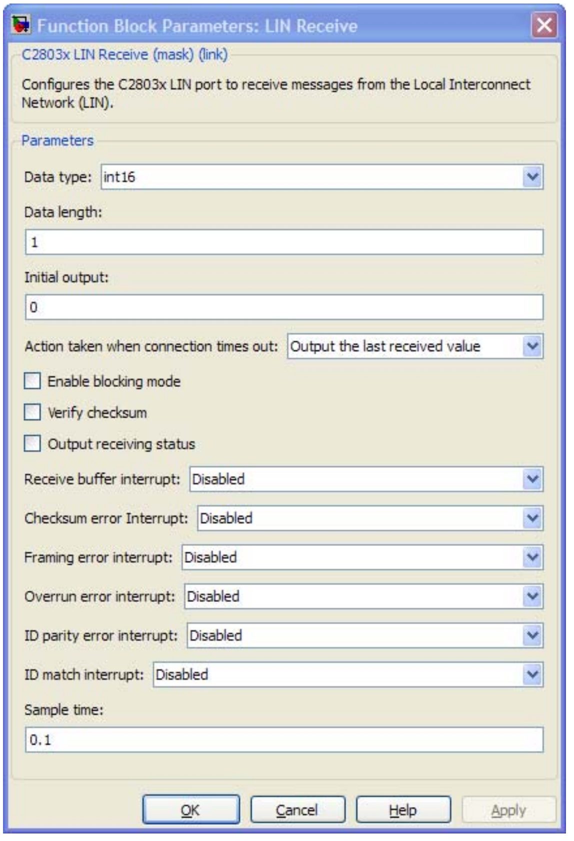
- **ID**: Set the value of the LIN ID for the LIN transmit node.
- **Tx ID Mask**: Set the value of the LIN ID mask for the LIN transmit node.
- **Data**: Connect this input to the data source.

For more information and examples, see:

- "Configuring LIN Communications"
- LIN-Based Control of PWM Duty Cycle (demo)

Note Many LIN-specific settings are located under **Peripherals** > **LIN** in the Target Preferences block for your model. Verify that these settings are correct for your application.

Dialog Box



Data type

Select the data type the LIN block outputs to the model. Available options are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. To interpret the data correctly, the data type and data length must match those of the data input to transmitting LIN node.

The default value is `int16`.

Data length

Set the length of the data the LIN block outputs to the model. This value is measured in multiples of the **Data type**. For example, if **Data type** is `int16` and **Data length** is `int16`, the LIN block outputs the data to the model in lengths of

1 x int16

If you set the **Data length** to a value greater than 1, the block outputs the data as vectors.

To interpret the data correctly, the data type and data length must match those of the data input to transmitting LIN node.

The default value is 1.

Note In a loopback configuration, the maximum data length cannot exceed 8 bytes. If the sum of the incoming and the outgoing data exceeds the hardware buffer length of the LIN module, the module discards incoming bytes of data.

Initial output

Set the initial value the DATA port outputs to the model before the LIN node has received any data.

The default value is `0`.

Action taken when connection times out

Specify what the LIN block outputs on the DATA port in response to a connection time-out. The choices are:

- `Output the last received value` — the DATA port outputs the last data value the LIN node received.
- `Output custom value` — the DATA port outputs the value defined by **Output value when connection times out**.

The default value is `Output the last received value`.

If the LIN node has not received data, and you set this parameter to `Output the last received value`, the DATA port outputs the **Initial output** value.

Output value when connection times out
Specify the custom value the DATA port outputs when **Action taken when connection times out** is set to `Output custom value` and a connection timeout occurs.

Enable blocking mode
If you enable (select) this checkbox, the target application stops and waits for the LIN node to receive data before continuing. If you disable this option, the application continues running and does not wait for data to arrive.

The default value is disabled (deselected).

Verify checksum
If you enable (select) this option, the LIN node verifies the checksum it receives.

The default value is disabled (deselected).

Output receiving status
Enabling (selecting) this checkbox adds a `status` output to the LIN Receive block, as shown in the following figure.

The `status` output reports the following values for each message the LIN node receives:

- `0`: No error.
- `-1`: A time-out occurred while the block was waiting to receive data.
- `-2`: Unable to receive.
- Other status values represent the highest 8 bits of the SCI Flags Register. Convert these values from decimal to binary.

Then determine the meaning of these values by referring to "Table 14. SCI Flags Register (SCIFLR) Field Descriptions" in *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2, available at the Texas Instruments Web site.

Receive buffer interrupt
If you enable this option, the SCI node generates an interrupt after it receives a complete frame. The default value is `Disabled`.

Checksum error interrupt
If you enable this option, the LIN block generates an interrupt when the incoming message contains an invalid checksum.

The default value is `Disabled`.

The TXRX Error Detector Checksum Calculator verifies checksums for incoming messages. With the classic LIN implementation, the checksum only covers the data fields. For LIN 2.0–compliant messages, the checksum includes both the ID field and the data fields. If you enable this option, the Checksum Calculator generates interrupts when it detects checksum errors, such as those caused by LIN message collisions.

Framing error interrupt
If you enable this option, the LIN module generates interrupts when framing errors occur.

The default value is `Disabled`.

Overrun error interrupt
If you enable this option, the LIN module generates interrupt when overrun errors occur.

The default value is `Disabled`.

ID parity error interrupt
If you enable this option, the LIN module generates an ID-Parity interrupt when it receives an invalid ID.

The default value is `Disabled`.

If you enable this option, also enable **Parity mode** in the Target Preferences block.

ID match interrupt

If you enable this option, the LIN module generates an interrupt when the LIN node validates the ID in messages it receives.

The default value is `Disabled`.

Sample time

Set the block's input sample time, T_s.

The default value is `0.1` seconds.

References

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.

See Also

C2803x LIN Transmit (block reference)

"LIN" on page 5-934 (block reference)

"Configuring LIN Communications"

LIN-Based Control of PWM Duty Cycle (demo)

C2803x LIN Transmit

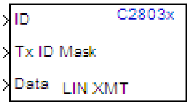
Purpose

Transmit data from target via serial communications interface (SCI) to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Description



LIN Transmit

The Local Interconnect Network (LIN) bus implements a serial communications protocol for distributed automotive and industrial applications. In particular, LIN serves low cost applications that do not require the bandwidth or robustness provided by the CAN protocol. For more information about LIN, see http://www.lin-subbus.org/.

The C2803x LIN Transmit block takes three inputs:

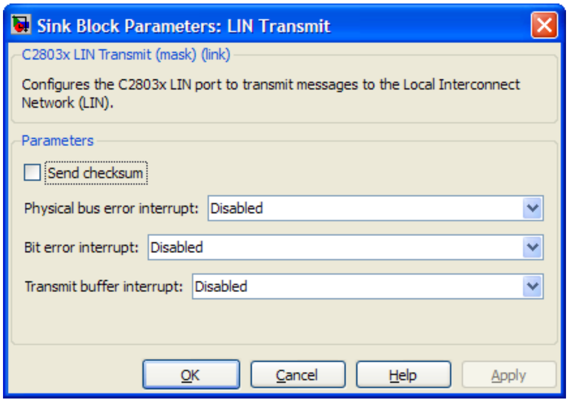
- **ID**: Set the value of the LIN ID for the LIN transmit node.
- **Tx ID Mask**: Set the value of the LIN ID mask for the LIN transmit node.
- **Data**: Connect this input to the data source.

For more information and examples, see:

- “Configuring LIN Communications”
- LIN-Based Control of PWM Duty Cycle (demo)

Note Many LIN-specific settings are located under **Peripherals** > **LIN** in the Target Preferences block for your model. Verify that these settings are correct for your application.

Dialog Box



Send checksum

Select this checkbox to include a checksum in the last data field of the checkbyte. LIN 2.0 implementations require this checksum.

The default value is unchecked (disabled).

Physical bus error interrupt

The LIN master node detects when the physical bus cannot convey a valid message. For example, if the bus had a short circuit to ground or to V_{BAT}. This raises a physical bus error flag in all of the LIN nodes on the network. If you enable **Physical bus error interrupt**, the LIN transmit node generates an interrupt in response to a physical bus error flag.

Bit error interrupt

If you enable this option, the LIN node compares the data it transmits and the data on the LIN bus.

The default value is `Disabled`.

The TXRX Error Detector Bit Monitor compares data bits on the LIN transmit (LINTX) and receive (LINRX) pins. If the data do not match, the Bit Monitor raises a bit-error flag. When you enable this option, the bit-error flag also produces a bit-error interrupt.

Transmit buffer interrupt

If you enable this option, the LIN node generates an interrupt while it is generating a checksum and setting the Transmitter buffer register ready flag.

The default value is `Disabled`.

References

For detailed information on the SCI module, see *TMS320F2803x Piccolo Local Interconnect Network (LIN) Module*, Literature Number SPRUGE2, available at the Texas Instruments Web site.

See Also

C2803x LIN Receive (block reference)

"LIN" on page 5-934 (block reference)

"Configuring LIN Communications"

LIN-Based Control of PWM Duty Cycle (demo)

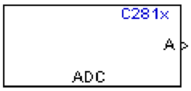
Purpose

Analog-to-digital converter (ADC)

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description



ADC

The C281x ADC block configures the C281x ADC to perform analog-to-digital conversion of signals connected to the selected ADC input pins. The ADC block outputs digital values representing the analog input signal and stores the converted values in the result register of your digital signal processor. You use this block to capture and digitize analog signals from external sources such as signal generators, frequency generators, or audio devices.

Triggering

The C281x ADC trigger mode depends on the internal setting of the source start-of-conversion (SOC) signal. In unsynchronized mode the ADC is usually triggered by software at the sample time intervals specified in the ADC block. For more information on configuring the specific parameters for this mode, see "Configuring Acquisition Window Width for ADC Blocks".

In synchronized mode, the Event (EV) Manager associated with the same module as the ADC triggers the ADC. In this case, the ADC is synchronized with the pulse width modulator (PWM) waveforms generated by the same EV unit via the **ADC Start Event** signal setting. The **ADC Start Event** is set in the C281x PWM block. See that block for information on the settings.

Note The ADC cannot be synchronized with the PWM if the ADC is in cascaded mode (see below).

Output

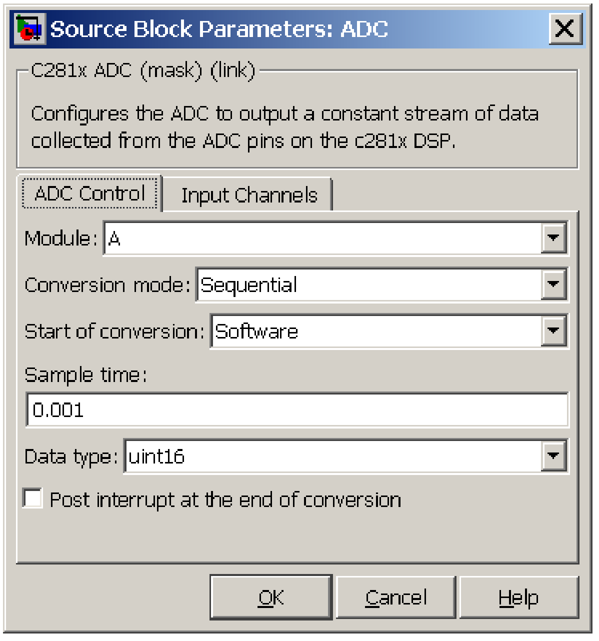
The output of the C281x ADC is a vector of `uint16` values. The output values are in the range 0 to 4095 because the C281x ADC is 12-bit converter.

Modes

The C281x ADC block supports ADC operation in dual and cascaded modes. In dual mode, either module A or module B can be used for the ADC block, and two ADC blocks are allowed in the model. In cascaded mode, both module A and module B are used for a single ADC block.

Dialog Box

ADC Control Pane



Module

Specify which DSP module to use:

- A — Displays the ADC channels in module A (ADCINA0 through ADCINA7).
- B — Displays the ADC channels in module B (ADCINB0 through ADCINB7).

- `A and B` — Displays the ADC channels in both modules A and B (ADCINA0 through ADCINA7 and ADCINB0 through ADCINB7)

Then, use the check boxes to select the desired ADC channels.

Conversion mode
Type of sampling to use for the signals:

- `Sequential` — Samples the selected channels sequentially
- `Simultaneous` — Samples the corresponding channels of modules A and B at the same time

Start of conversion
Specify the type of signal that triggers the conversion:

- `Software` — Signal from software
- `EVA` — Signal from Event Manager A (only for Module A)
- `EVB` — Signal from Event Manager B (only for Module B)
- `External` — Signal from external hardware

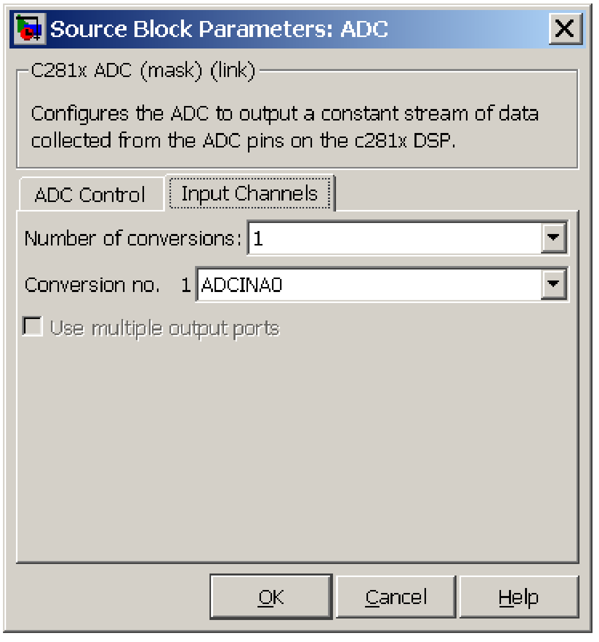
Sample time
Time in seconds between consecutive sets of samples that are converted for the selected ADC channel(s). This is the rate at which values are read from the result registers. See"Scheduling and Timing" for more information on timing. To execute this block asynchronously, set **Sample Time** to `-1`, check the **Post interrupt at the end of conversion** box, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

To set different sample times for different groups of ADC channels, you must add separate C281x ADC blocks to your model and set the desired sample times for each block.

Data type
Date type of the output data. Valid data types are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

Post interrupt at the end of conversion
Check this check box to post an asynchronous interrupt at the end of each conversion. The interrupt is always posted at the end of conversion.

Input Channels Pane



Number of conversions
Number of ADC channels to use for analog-to-digital conversions.

Conversion no.
Specific ADC channel to associate with each conversion number.

In oversampling mode, a signal at a given ADC channel can be sampled multiple times during a single conversion sequence. To oversample, specify the same channel for more than one conversion. Converted samples are output as a single vector.

Use multiple output ports

If more than one ADC channel is used for conversion, you can use separate ports for each output and show the output ports on the block. If you use more than one channel and do not use multiple output ports, the data is output in a single vector.

See Also

C281x PWM

C281x Hardware Interrupt

"ADC" on page 5-883

C281x CAP

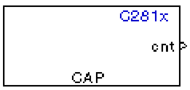
Purpose

Receive and log capture input pin transitions

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description



CAP

The C281x CAP block sets parameters for the capture units (CAPs) of the Event Manager (EV) module. The capture units log transitions detected on the capture unit pins by recording the times of these transitions into a two-level deep FIFO stack. You can set the capture unit pins to detect rising edge, falling edge, either type of transition, or no transition.

The C281x chip has six capture units — three associated with each EV module. Capture units 1, 2, and 3 are associated with EVA and capture units 4, 5, and 6 are associated with EVB. Each capture unit is associated with a capture input pin.

Each group of EV module capture units can use one of two general-purpose (GP) timers on the target board. EVA capture units can use GP timer 1 or 2. EVB capture units can use GP timer 3 or 4. When a transition occurs, the module stores the value of the selected timer in the two-level deep FIFO stack.

The C281x CAP module shares GP Timers with other C281 blocks. For more information and guidance on sharing timers, see "Sharing General Purpose Timers between C281x Peripherals".

Note You can have up to two C281x CAP blocks in any one model—one block for each EV module.

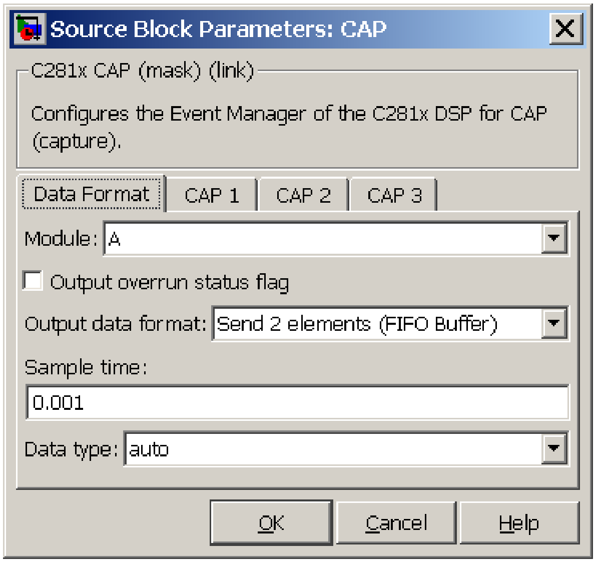
Outputs

This block has up to two outputs: a `cnt` (count) output and an optional, FIFO status `flag` output. The `cnt` output increments each time a transition of the selected type occurs. The status flag outputs are

- 0 — The FIFO is empty. Either no captures have occurred or the previously stored captures have been read from the stack. (The binary version of this flag is 00.)
- 1 — The FIFO has one entry in the top register of the stack. (The binary version of this flag is 01.)
- 2 — The FIFO has two entries in the stack registers. (The binary version of this flag is 10.)
- 3 — The FIFO has two entries in the stack registers and one or more captured values have been lost. This occurs because another capture occurred before the FIFO stack was read. The new value is placed in the bottom register. The bottom register value is pushed to the top of the stack and the top value is pushed out of the stack. (The binary version of this flag is 11.)

Dialog Box

Data Format Pane



Module

Select the Event Manager (EV) module to use:

- `A` — Use CAPs 1, 2, and 3.
- `B` — Use CAPs 4, 5, and 6.

Output overrun status flag

Select to output the status of the elements in the FIFO. The data type of the status flag is uint16.

Send data format

The type of data to output:

- `Send 2 elements (FIFO Buffer)` — Sends the latest two values. The output is updated when there are two elements in the FIFO, which is indicated by bit 13 or 11 or 9 being sent (CAP x FIFO). If the CAP is polled when fewer than two elements are captures, old values are repeated. The CAP registers are read as follows:

  1. The CAP x FIFO status bits are read and the value is stored in the status flag.
  2. The top value of the FIFO is read and stored in the output at index 0.
  3. The new top value of the FIFO (the previously stored bottom stack value) is read and stored in the output at index 1.

- `Send 1 element (oldest)` — Sends the older of the two most recent values. The output is updated when there is at least one element in the FIFO, which is indicated by any of the bits 13:12, or 11:10, or 9:8 being sent. The CAP registers are read as follows:

  4. The CAP x FIFO status bits are read and the value is stored in the status flag.
  5. The top value of the FIFO is read and stored in the output.

- `Send 1 element (latest)` — Sends the most recent value. The output is updated when there is at least one element in the

FIFO, which is indicated by any of the bits 13:12, or 11:10, or 9:8 being sent. The CAP registers are read as follows:

6. The CAP x FIFO status bits are read and the value is stored in the status flag.
7. If there are two entries in the FIFO, the bottom value is read and stored in the output. If there is only one entry in the FIFO, the top value is read and stored in the output.

Sample time

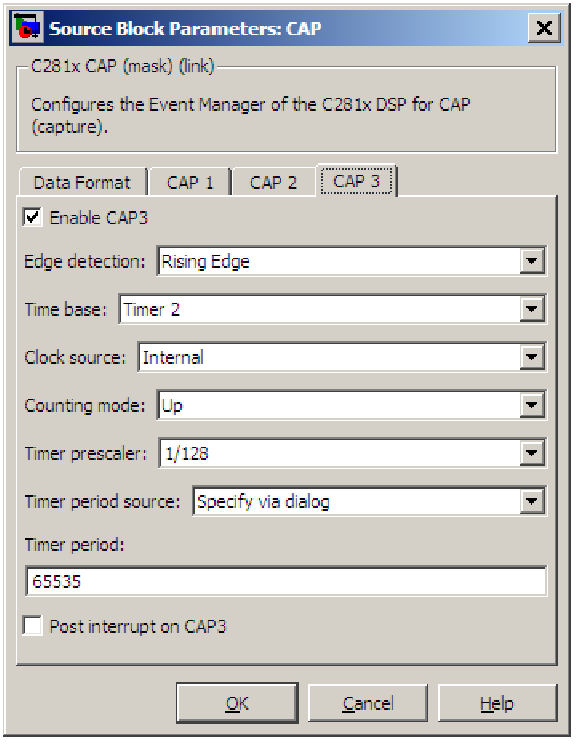
Time between outputs from the FIFO. If new data is not available, the previous data is sent.

Data type

Data type of the output data. Available options are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `boolean`. The `auto` option uses the data type of a connected block that outputs data to this block. If this block does not receive any input, `auto` sets the data type to `double`.

Note The output of the C281x CAP block can be vectorized.

CAP Panes



The CAP panes set parameters for individual CAPs. The particular CAP affected by a CAP pane depends on the EV module you selected:

- **CAP1** controls CAP 1 or CAP 4, for EV module A or B, respectively.
- **CAP2** controls CAP 2 or CAP 5, for EV module A or B, respectively.
- **CAP3** controls CAP 3 or CAP 6, for EV module A or B, respectively.

Enable CAP
: Select to use the specified capture unit pin.

Edge Detection

Type of transition detection to use for this CAP. Available types are `Rising Edge`, `Falling Edge`, `Both Edges`, and `No transition`.

Time Base

Select which target board GP timer the CAP uses as a time base. CAPs 1, 2, and 3 can use `Timer 1` or `Timer 2`. CAPs 4, 5, and 6 can use `Timer 3` or `Timer 4`.

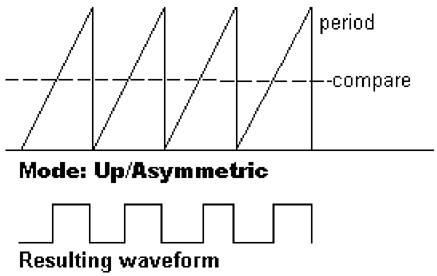
Clock source

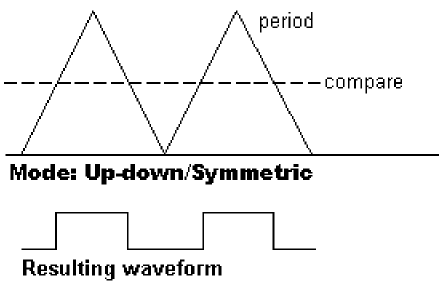
This option is available only for the CAP 3 pane. You can select `Internal` to use the internal time base. Also configure the **Counting mode**, **Timer prescaler**, and **Timer period source** for the internal time base.

Select `QEP circuit` to generate the input clock from the quadrature encoder pulse (QEP) submodule.

Counting mode

Select `Up` to generate an asymmetrical waveform output, or `Up-down` to generate a symmetrical waveform output, as shown in the following illustration.





When you specify the **Counting mode** as `Up` (asymmetric) the waveform:

- Starts low
- Goes high when the rising period counter value matches the **Compare value**
- Goes low at the end of the period

When you specify the **Counting mode** as `Up-down` (symmetric) the waveform:

- Starts low
- Goes high when the increasing period counter value matches the **Compare value**

- Goes low when the decreasing period counter value matches the **Compare value**

Counting mode becomes unavailable when you set **Clock source** to `QEP circuit`.

Timer Prescaler

Clock divider factor by which to prescale the selected GP timer to produce the desired timer counting rate. Available options are `none`, `1/2`, `1/4`, `1/8`, `1/16`, `1/32`, `1/64`, and `1/128`. The following table shows the rates that result from selecting each option.

Scaling	Resulting Rate (μs)
none	0.01334
1/2	0.02668
1/4	0.05336
1/8	0.10672
1/16	0.21344
1/32	0.42688
1/64	0.85376
1/128	1.70752

Note These rates assume a 75 MHz input clock.

Timer period source

Select `Specify via dialog` to enable the **Timer period** parameter. Select `Input port` to create a block input, **T1**, that accepts the timer period value.

Timer period

Set the length of the timer period in clock cycles. Enter a value from `0` to `65535`. The value defaults to `65535`.

If you know the length of a clock cycle, you can easily calculate how many clock cycles to set for the timer period. The following calculation determines the length of one clock cycle:

$$Sysclk(150MHz) \rightarrow HISPCLK(1/2) \rightarrow InputClock\,\Pr escaler(1/128)$$

In this calculation, you divide the System clock frequency of 150 MHz by the high-speed clock prescaler of 2. Then, you divide the resulting value by the timer control input clock prescaler, 128. The resulting frequency is 0.586 MHz. Thus, one clock cycle is 1/.586 MHz, which is 1.706 μs.

Post interrupt on CAP
Check this check box to post an asynchronous interrupt on CAP.

See Also

C281x Hardware Interrupt

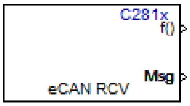
Purpose

Enhanced Control Area Network receive mailbox

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description



eCAN Receive

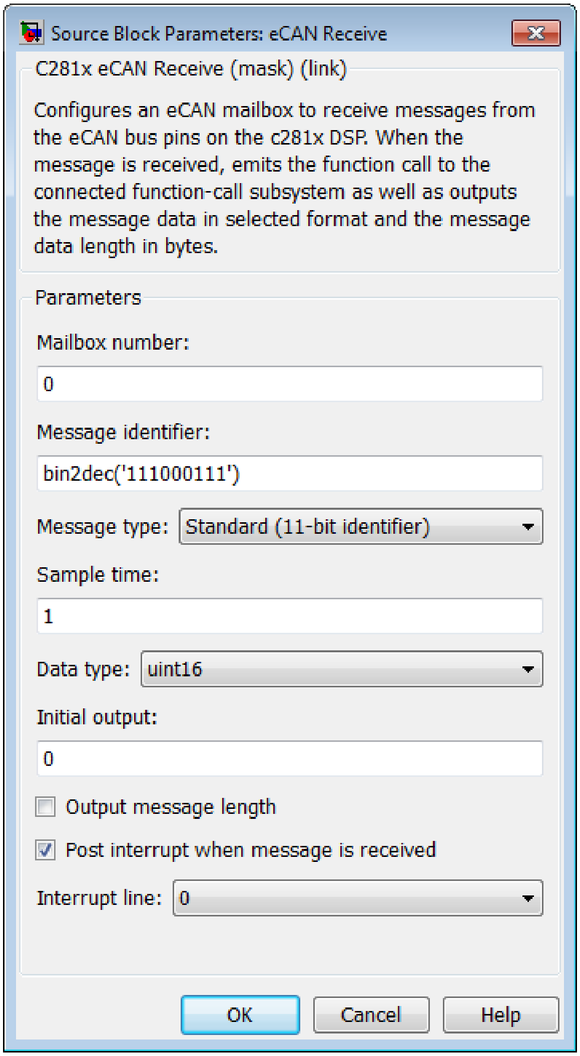
Use the C281x enhanced Control Area Network (eCAN) Receive block to receive eCAN messages through an eCAN mailbox. The eCAN module on the DSP chip provides serial communication capability and has 32 mailboxes configurable for receive or transmit. The C281x supports eCAN data frames in standard or extended format.

The C281x eCAN Receive block has up to three output ports.

- **f0** outputs a function call when the block receives a new message. Connect a function call subsystem to this port.
- **Msg** outputs the message data as a vector. The vector is always 8 bytes long. Use **Data type** to and is composed of elements of the data type.
- **len** outputs the length of the eCAN message. Select **Output message length** to create this output.

To use the eCAN Receive block with the eCAN Pack block in the `canmsglib`, set **Data type** to `CAN_MESSAGE_TYPE`.

Dialog Box


Source Block Parameters: eCAN Receive
C281x eCAN Receive (mask) (link)
Configures an eCAN mailbox to receive messages from the eCAN bus pins on the c281x DSP. When the message is received, emits the function call to the connected function-call subsystem as well as outputs the message data in selected format and the message data length in bytes.
Parameters
Mailbox number:
0
Message identifier:
bin2dec('111000111')
Message type:
Standard (11-bit identifier)
Sample time:
1
Data type:
uint16
Initial output:
0
Output message length
Post interrupt when message is received
Interrupt line:
0
OK
Cancel
Help

Mailbox number

Unique number between 0 and 15 for standard or between 0 and 31 for enhanced CAN mode. It refers to a mailbox area in RAM. In standard mode, the mailbox number determines priority.

Message identifier

Identifier of length 11 bits for standard frame size or length 29 bits for extended frame size in decimal, binary, or hex. If in binary or hex, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry. The message identifier is associated with a receive mailbox. Only messages that match the mailbox message identifier are accepted into it.

Message type

Select `Standard (11-bit identifier)` or `Extended (29-bit identifier)`.

Sample time

Frequency with which the mailbox is polled to determine if a new message has been received. A new message causes a function call to be emitted from the mailbox. If you want to update the message output only when a new message arrives, then the block needs to be executed asynchronously. To execute this block asynchronously, set **Sample Time** to `-1`, check the **Post interrupt when message is received** box, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

Note For information about setting the timing parameters of the CAN module, see "Configuring Timing Parameters for CAN Blocks".

Data type

Select one of the following options:

- `uint8` (vector length = 8 elements)

- `uint16` (vector length = 4 elements)
- `uint32` (vector length = 2 elements)
- `CAN_MESSAGE_TYPE` (Select this option to use the eCAN receive block with the CAN Unpack block.)

The length of the vector for the received message is at most 8 bytes. If the message is less than 8 bytes, the data buffer bytes are right-aligned in the output. The data are unpacked as follows using the data buffer, which is 8 bytes.

For `uint8` data, eCAN Receive reads each unit of 8 bytes in the registers, and outputs 8-bit data to 8 elements (using the lower part of the 16-bit memory):

```
Output[0] = data_buffer[0];
Output[1] = data_buffer[1];
Output[2] = data_buffer[2];
Output[3] = data_buffer[3];
Output[4] = data_buffer[4];
Output[5] = data_buffer[5];
Output[6] = data_buffer[6];
Output[7] = data_buffer[7];
```

For `uint16` data,

```
Output[0] = data_buffer[1..0];
Output[1] = data_buffer[3..2];
Output[2] = data_buffer[5..4];
Output[3] = data_buffer[7..6];
```

For `uint32` data,

```
Output[0] = data_buffer[3..0];
Output[1] = data_buffer[7..4];
```

For example, if the received message has two bytes:

```
data_buffer[0] = 0x21
data_buffer[1] = 0x43
```

The `uint16` output would be:

```
Output[0] = 0x4321
Output[1] = 0x0000
Output[2] = 0x0000
Output[3] = 0x0000
```

When you select CAN_MESSAGE_TYPE, the block outputs the following struct data (defined in can_message.h):

```
struct {

  /* Is Extended frame */
  uint8_T Extended;

  /* Length */
  uint8_T Length;

  /* RTR */
  uint8_T Remote;

  /* Error */
  uint8_T Error;

  /* CAN ID */
  uint32_T ID;

  /*
  TIMESTAMP_NOT_REQUIRED is a macro that will be defined by Target teams
  PIL, xPC if they do not require the timestamp field during code
  generation. By default, timestamp is defined. If the targets do not require
  the timestamp field, they should define the macro TIMESTAMP_NOT_REQUIRED before
  including this header file for code generation.
  */
```

```
    #ifndef TIMESTAMP_NOT_REQUIRED
        /* Timestamp */
        double Timestamp;
    #endif

    /* Data field */
    uint8_T Data[8];

};
```

Output message length
Select to output the message length in bytes to the third output port. If not selected, the block has only two output ports.

Post interrupt when message is received
Select this check box to post an asynchronous interrupt when a message is received.

Interrupt line
Select the interrupt line the asynchronous interrupt uses. This action sets bit 2 (GIL) in the Global Interrupt Mask Register (CANGIM):

- 1 maps the global interrupts to the ECAN1INT line.
- 0 maps the global interrupts to the ECAN0INT line.

References

For detailed information on the eCAN module, see *TMS320F28x DSP Enhanced Control Area Network (eCAN) Reference Guide*, Literature Number SPRU074A, available at the Texas Instruments Web site.

See Also

C281x eCAN Transmit,

C281x Hardware Interrupt

"eCAN_A, eCAN_B" on page 5-886

Purpose

Enhanced Control Area Network transmit mailbox

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description

eCAN Transmit

The C281x enhanced Control Area Network (eCAN) Transmit block generates source code for transmitting eCAN messages through an eCAN mailbox. The eCAN module on the DSP chip provides serial communication capability and has 32 mailboxes configurable for receive or transmit. The C28x supports eCAN data frames in standard or extended format.

Note Fixed-point inputs are not supported for this block.

Data Vectors

The length of the vector for each transmitted mailbox message is 8 bytes. Input data are always right-aligned in the message data buffer. Only `uint16` (vector length = 4 elements) or `uint32` (vector length = 2 elements) data are accepted. The following examples show how the different types of input data are aligned in the data buffer

For input of type `uint32`,

```
inputdata [0] = 0x12345678
```

the data buffer is:

```
data buffer[0] = 0x78
data buffer[1] = 0x56
data buffer[2] = 0x34
data buffer[3] = 0x12
data buffer[4] = 0x00
data buffer[5] = 0x00
data buffer[6] = 0x00
data buffer[7] = 0x00
```

For input of type `uint16`,

```
inputdata [0] = 0x1234
```

the data buffer is:

```
data buffer[0] = 0x34
data buffer[1] = 0x12
data buffer[2] = 0x00
data buffer[3] = 0x00
data buffer[4] = 0x00
data buffer[5] = 0x00
data buffer[6] = 0x00
data buffer[7] = 0x00
```

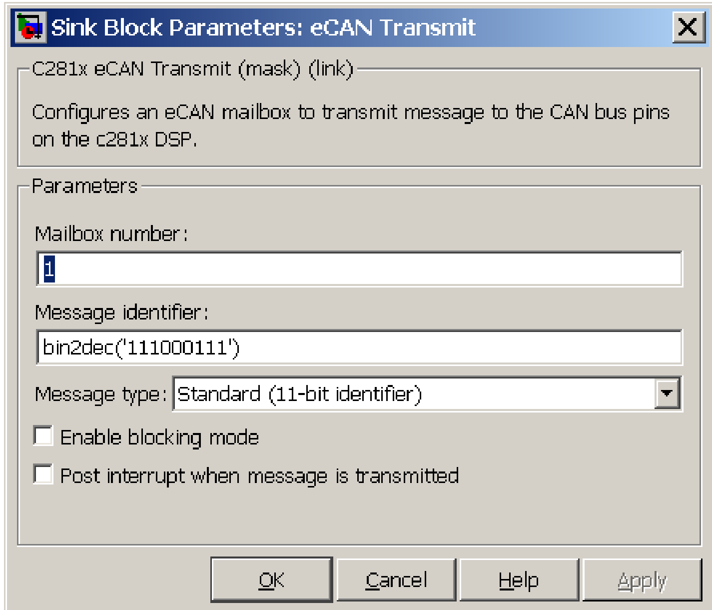
For input of type `uint16[2]`, which is a two-element vector,

```
inputdata [0] = 0x1234
inputdata [1] = 0x5678
```

the data buffer is:

```
data buffer[0] = 0x34
data buffer[1] = 0x12
data buffer[2] = 0x78
data buffer[3] = 0x56
data buffer[4] = 0x00
data buffer[5] = 0x00
data buffer[6] = 0x00
data buffer[7] = 0x00
```

Dialog Box



Mailbox number

Unique number between 0 and 15 for standard or between 0 and 31 for enhanced CAN mode. It refers to a mailbox area in RAM. In standard mode, the mailbox number determines priority.

Message identifier

Identifier of length 11 bits for standard frame size or length 29 bits for extended frame size in decimal, binary, or hex. If in binary or hex, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry. The message identifier is coded into a message that is sent to the CAN bus.

Message type

Select `Standard (11-bit identifier)` or `Extended (29-bit identifier)`.

Enable blocking mode
If selected, the CAN block code waits indefinitely for a transmit (XMT) acknowledge. If cleared, the CAN block code does not wait for a transmit (XMT) acknowledge, which is useful when the hardware might fail to acknowledge transmissions.

Post interrupt when message is transmitted
If selected, an asynchronous interrupt is posted when data is transmitted.

Interrupt Line
Select the interrupt line the asynchronous interrupt uses, 0 or 1.

Note For information about setting the timing parameters of the CAN module, see "Configuring Timing Parameters for CAN Blocks".

References

For detailed information on the eCAN module, see *TMS320F28x DSP Enhanced Control Area Network (eCAN) Reference Guide*, Literature Number SPRU074A, available at the Texas Instruments Web site.

See Also

C281x eCAN Receive

"eCAN_A, eCAN_B" on page 5-886

C281x GPIO Digital Input

Purpose

General-purpose I/O pins for digital input

Library

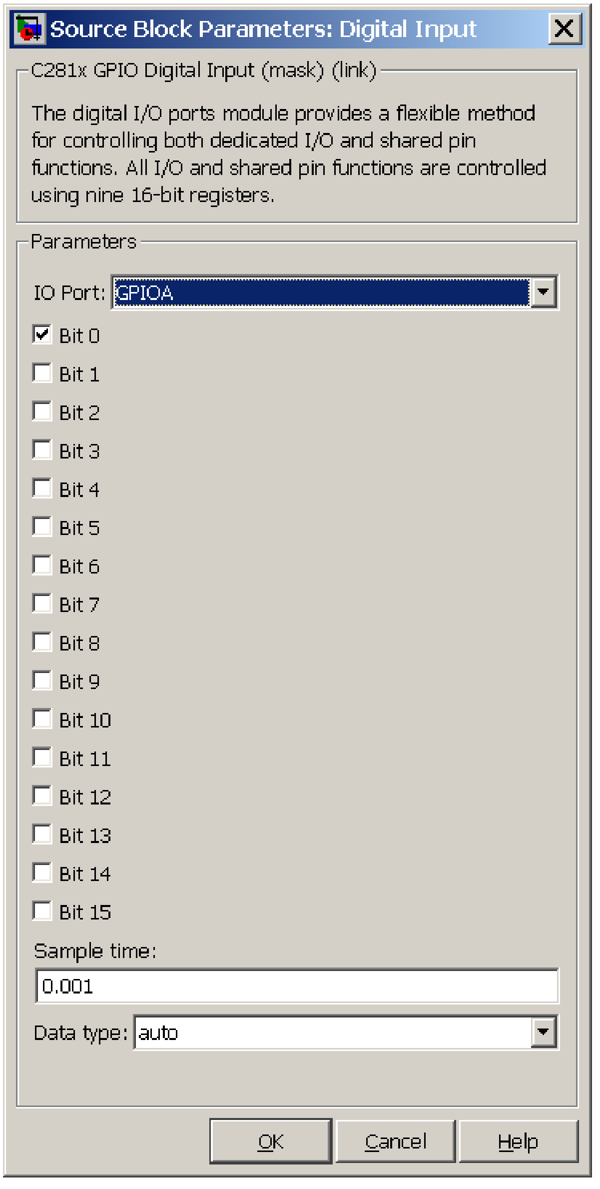
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description

This block configures the general-purpose I/O (GPIO) registers that control the GPIO shared pins for digital input. Each I/O port has one MUX register, which is used to select peripheral operation or digital I/O operation.

Note To avoid losing any new settings, click **Apply** before changing the **IO Port** parameter.

Dialog Box



Source Block Parameters: Digital Input
C281x GPIO Digital Input (mask) (link)
The digital I/O ports module provides a flexible method for controlling both dedicated I/O and shared pin functions. All I/O and shared pin functions are controlled using nine 16-bit registers.
Parameters
IO Port: GPIOA
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
Bit 8
Bit 9
Bit 10
Bit 11
Bit 12
Bit 13
Bit 14
Bit 15
Sample time:
0.001
Data type: auto
OK
Cancel
Help

IO Port

Select the input/output port to use: `GPIOPA`, `GPIOPB`, `GPIOPD`, `GPIOPE`, `GPIOPF`, or `GPIOPG` and select the I/O Port bits to enable for digital input. (There is no GPIOPC port on the C281x.) If you select multiple bits, vector input is expected. Cleared bits are available for peripheral functionality. Multiple GPIO DI blocks cannot share the same I/O port.

Note The input function of the digital I/O and the input path to the related peripheral are always enabled on the board. If you configure a pin as digital I/O, the corresponding peripheral function cannot be used.

The following tables show the shared pins.

GPIO A MUX

Bit	Peripheral Name (Bit =1)	GPIO Name (Bit = 0)
0	PWM1	GPIOA0
1	PWM2	GPIOA1
2	PWM3	GPIOA2
3	PWM4	GPIOA3
4	PWM5	GPIOA4
5	PWM6	GPIOA5
8	QEP1/CAP1	GPIOA8
9	QEP2/CAP2	GPIOA9
10	CAP3	GPIOA10

GPIO B MUX

Bit	Peripheral Name (Bit =1)	GPIO Name (Bit = 0)
0	PWM7	GPIOB0
1	PWM8	GPIOB1
2	PWM9	GPIOB2
3	PWM10	GPIOB3
4	PWM11	GPIOB4
5	PWM12	GPIOB5
8	QEP3/CAP4	GPIOB8
9	QEP4/CAP5	GPIOB9
10	CAP6	GPIOB10

Sample time

Time interval, in seconds, between consecutive input from the pins.

Data type

Data type of the data to obtain from the GPIO pins. The data is read as 16-bit integer data and then cast to the selected data type. Valid data types are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` or `boolean`.

Note The width of the vectorized data output by this block is determined by the number of bits selected in the **Block Parameters** dialog box.

See Also

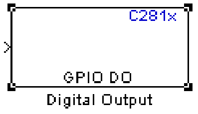
C281x GPIO Digital Output

"GPIO" on page 5-911

Purpose General-purpose I/O pins for digital output

Library Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

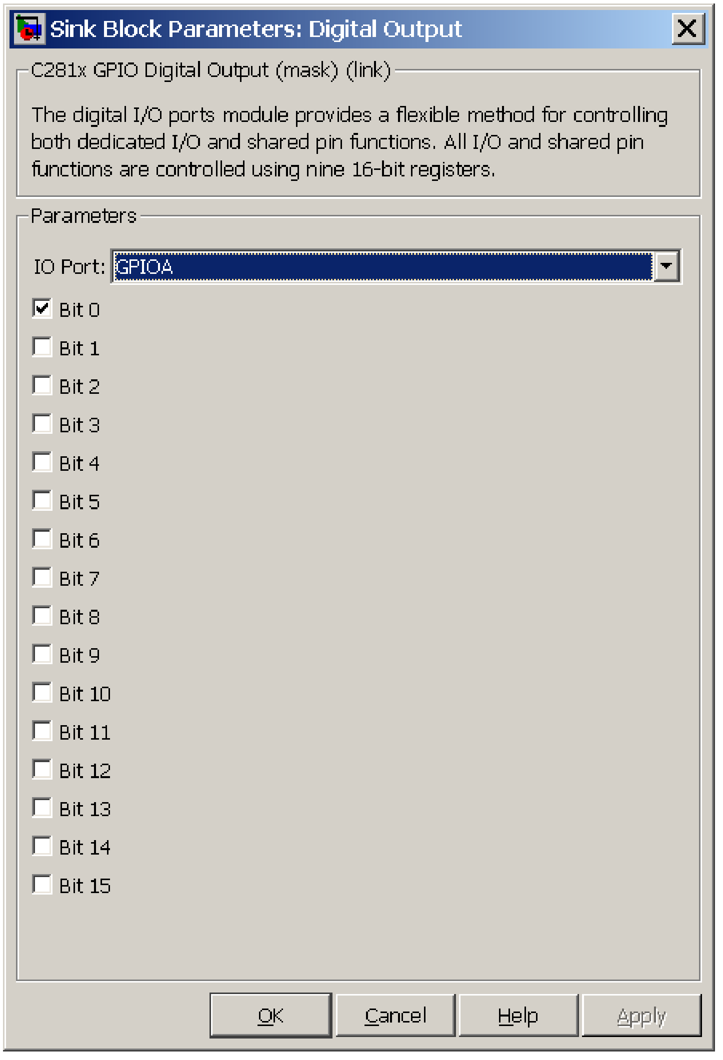
Description This block configures the general-purpose I/O (GPIO) registers that control the GPIO shared pins for digital output. Each I/O port has one MUX register, which is used to select peripheral operation or digital I/O operation.



Note Fixed-point inputs are not supported for this block.

Note To avoid losing any new settings, click **Apply** before changing the **IO Port** parameter.

Dialog Box



IO Port

Select the input/output port to use: `GPIOPA`, `GPIOPB`, `GPIOPD`, `GPIOPE`, `GPIOPF`, or `GPIOPG` and select the I/O Port bits to enable for digital input. (There is no GPIOPC port on the C281x.) If you select multiple bits, vector input is expected. Cleared bits are available for peripheral functionality. Multiple GPIO DO blocks cannot share the same I/O port.

Note The input function of the digital I/O and the input path to the related peripheral are always enabled on the board. If you configure a pin as digital I/O, the corresponding peripheral function cannot be used.

The following tables show the shared pins.

GPIO A MUX

Bit	Peripheral Name (Bit =1)	GPIO Name (Bit = 0)
0	PWM1	GPIOA0
1	PWM2	GPIOA1
2	PWM3	GPIOA2
3	PWM4	GPIOA3
4	PWM5	GPIOA4
5	PWM6	GPIOA5
8	QEP1/CAP1	GPIOA8
9	QEP2/CAP2	GPIOA9
10	CAP3	GPIOA10

GPIO B MUX

Bit	Peripheral Name (Bit =1)	GPIO Name (Bit = 0)
0	PWM7	GPIOB0
1	PWM8	GPIOB1
2	PWM9	GPIOB2
3	PWM10	GPIOB3
4	PWM11	GPIOB4
5	PWM12	GPIOB5
8	QEP3/CAP4	GPIOB8
9	QEP4/CAP5	GPIOB9
10	CAP6	GPIOB10

See Also

C281x GPIO Digital Input

"GPIO" on page 5-911

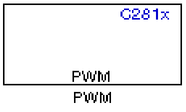
Purpose

Pulse width modulators (PWMs)

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description



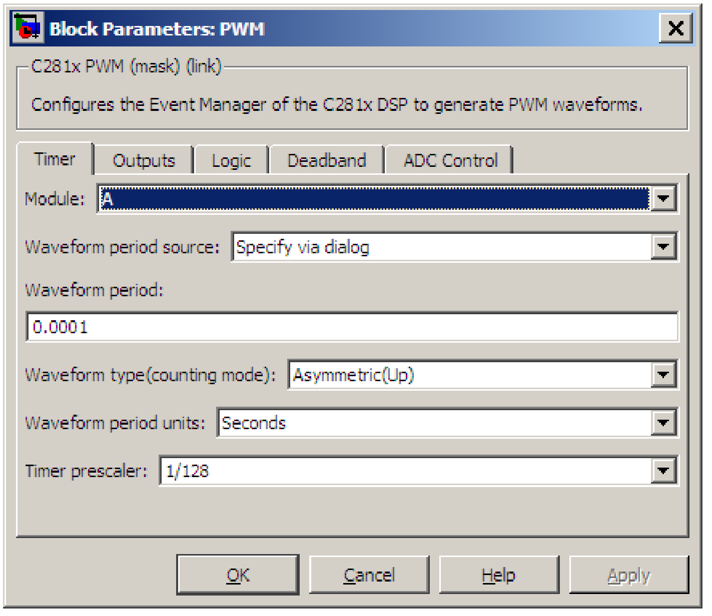
F2812 DSPs include a suite of pulse width modulators (PWMs) used to generate various signals. This block provides options to set the A or B module Event Managers to generate the waveforms you require. The twelve PWMs are configured in six pairs, with three pairs in each module.

The C281x PWM module shares GP Timers with other C281 blocks. For more information and guidance on sharing timers, see "Sharing General Purpose Timers between C281x Peripherals".

Note All inputs to the C281x PWM block must be scalar values.

Dialog Box

Timer Pane



Module

Specify which target PWM pairs to use:

- A — Displays the PWMs in module A (PWM1/PWM2, PWM3/PWM4, and PWM5/PWM6).
- B — Displays the PWMs in module B (PWM7/PWM8, PWM9/PWM10, and PWM11/PWM12).

Note PWMs in module A use Event Manager A, Timer 1, and PWMs in module B use Event Manager B, Timer 3.

Waveform period source

Source from which the waveform period value is obtained. Select `Specify via dialog` to enter the value in **Waveform period** or select `Input port` to use a value from the input port.

Note All inputs to the C281x PWM block must be scalar values.

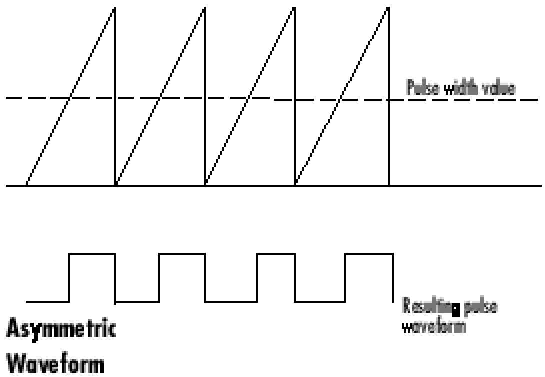
Waveform period

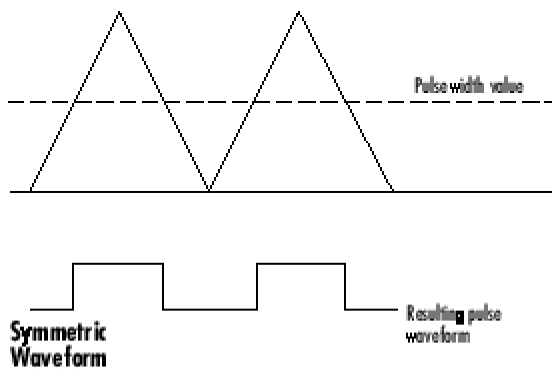
Period of the PWM waveform measured in clock cycles or in seconds, as specified in the **Waveform period units**.

Note The term *clock cycles* refers to the high-speed peripheral clock on the F2812 chip. This clock is 75 MHz by default because the high-speed peripheral clock prescaler is set to 2 (150 MHz/2).

Waveform type (counting mode)

Type of waveform to be generated by the PWM pair. The F2812 PWMs can generate two types of waveforms: `Asymmetric(Up)` and `Symmetric(Up-down)`. The following illustration shows the difference between the two types of waveforms.





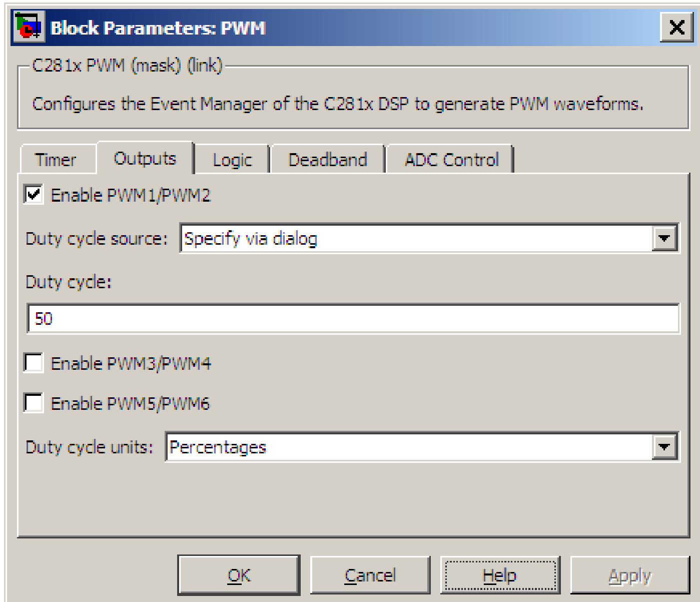
Waveform period units

Units in which to measure the waveform period. Options are `Clock cycles`, which refer to the high-speed peripheral clock on the F2812 chip (75 MHz), or `Seconds`. Changing these units changes the **Waveform period** value and the **Duty cycle** value and **Duty cycle units** selection.

Timer prescaler

Divide the clock input to produce the desired timer counting rate.

Outputs Pane



Enable PWM#/PWM#

Check to activate the PWM pair. PWM1/PWM2 are activated via the Output 1 pane, PWM3/PWM4 are on Output 2, and PWM5/PWM6 are on Output 3.

Duty cycle source

Select `Specify via dialog` to use the dialog box to enter a **Duty cycle** value for the pair of PWM outputs. Select `Input port` to use the input port, **W#**, to enter a **Duty cycle** value for the pair of PWM outputs.

The input port **W1** corresponds to PWM1/PWM2. **W2** corresponds to PWM3/PWM4. **W3** corresponds to PWM5/6.

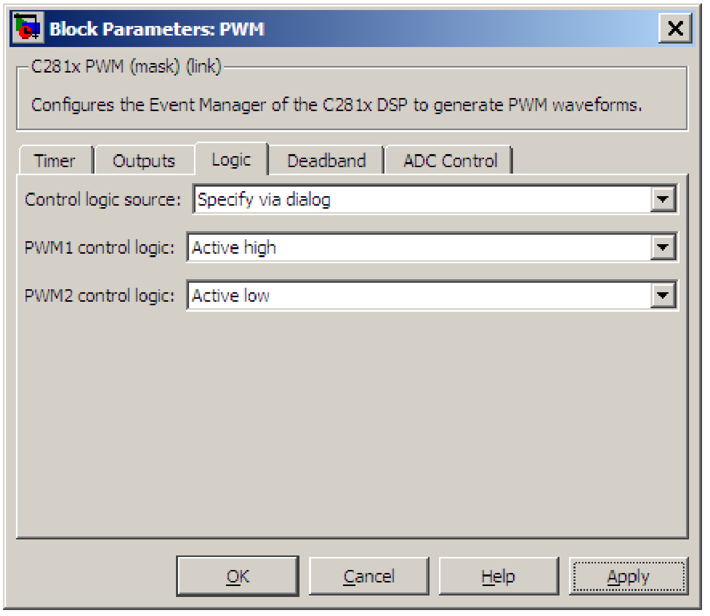
Note All inputs to the C281x PWM block must be scalar values.

Duty cycle
Set the ratio of the PWM waveform pulse duration to the PWM **Waveform period**.

Duty cycle units
Units for the duty cycle. Valid choices are `Clock cycles` and `Percentages`. Changing these units changes the **Duty cycle** value, and the **Waveform period** value and **Waveform period units** selection.

Note Using percentages can cause some additional computation time in generated code. This may or may not be noticeable in your application.

Logic Pane



Control logic source

Configure the control logic for all PWMs enabled on the Outputs tab. Valid settings are `Specify via dialog` (default setting) or to `Input port`.

`Specify via Dialog` enables **PWM control logic** settings for each PWM output:

- `Forced high` causes the pulse value to be high.

  `Active high` causes the pulse value to go from low to high.

  `Active low` causes the pulse value to go from high to low.

  `Forced low` causes the pulse value to be low.

`Input port` adds an input port to the PWM block for setting the C2000 ACTRx register. Each PWM uses 2 bits to set the following options:

- 00 Forced Low
- 01 Active Low
- 10 Active High
- 11 Forced High

Bits 11–0 of the 16–bit Compare Action Control Registers for module A control PWM1-6

Bits 11–0 of the 16–bit Compare Action Control Registers for module B control PWM1-6

For example: If a decimal value of 3222 is read at the input port while using PWM module A, the following PWM settings will be honored:

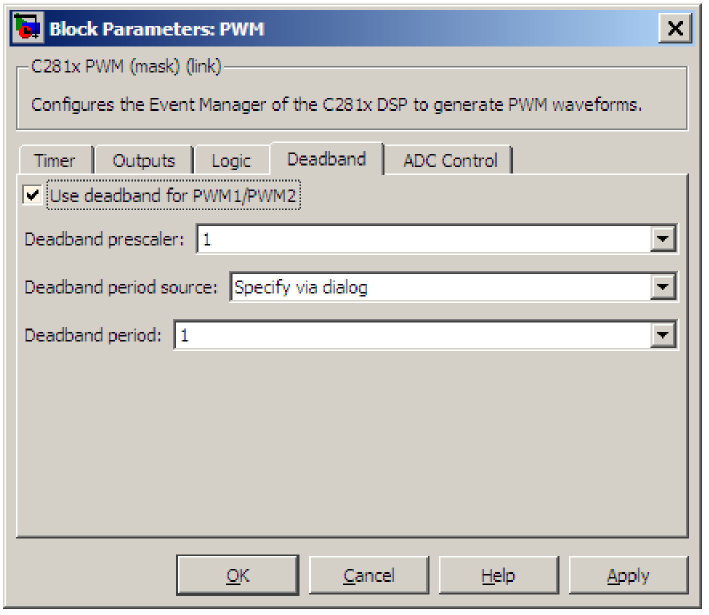
3222 = 0C96h = 110010010110b

So that:

- PW1: Active High
- PW2: Active Low
- PW3: Active Low
- PW4: Active High
- PW5: Forced Low
- PW6: Forced High

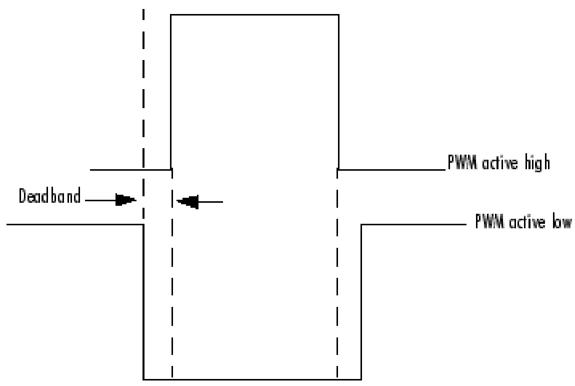
For more information, see the section on Compare Action Control Registers (ACTRA and ACTRB) in the Texas Instruments™ document "TMS320x281x DSP Event Manager (EV) Reference Guide", literature number SPRU065.

Deadband Pane



Use deadband for PWM#/PWM#

Enables a deadband area of no signal overlap at the beginning of particular PWM pair signals. The following figure shows the deadband area.



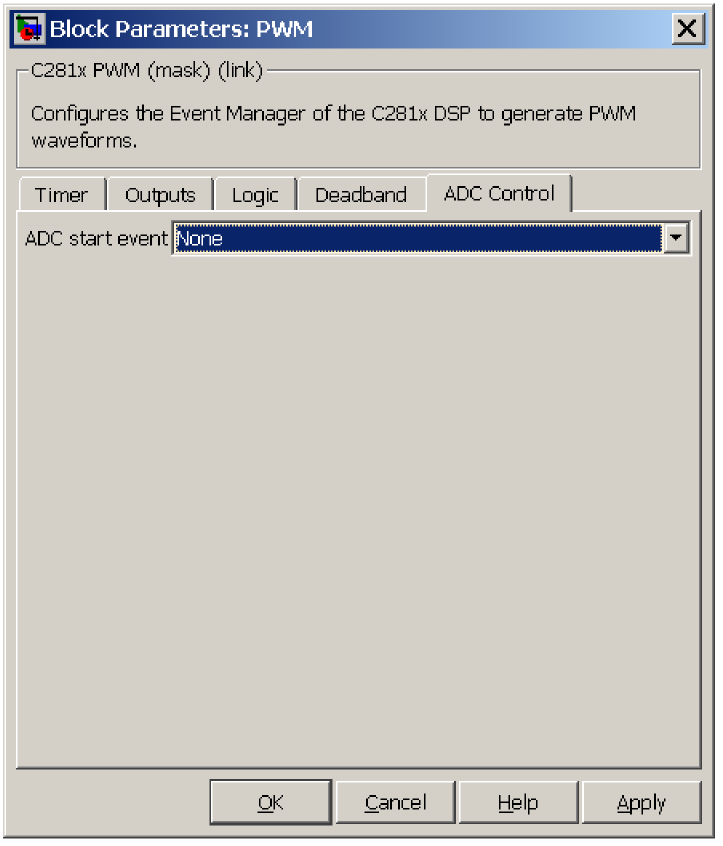
Deadband prescaler
Number of clock cycles, which, when multiplied by the Deadband period, determines the size of the deadband. Selectable values are 1, 2, 4, 8, 16, and 32.

Deadband period source
Source from which the deadband period is obtained. Select `Specify via dialog` to enter the values in the **Deadband period** field or select `Input port` to use a value, in clock cycles, from the input port.

Note All inputs to the C281x PWM block must be scalar values.

Deadband period
Value that, when multiplied by the Deadband prescaler, determines the size of the deadband. Selectable values are from 1 to 15.

ADC Control Pane



ADC start event

Controls whether this PWM and ADC associated with the same EV module are synchronized. Select `None` for no synchronization or select an event to generate the source start-of-conversion (SOC) signal for the associated ADC.

- `None` — The ADC and PWM are not synchronized. The EV does not generate an SOC signal and the ADC is triggered by software (that is, the A/D conversion occurs when the ADC block is executed in the software).

- `Underflow interrupt` — The EV generates an SOC signal for the ADC associated with the same EV module when the board's general-purpose (GP) timer counter reaches a hexadecimal value of FFFF.
- `Period interrupt` — The EV generates an SOC signal for the ADC associated with the same EV module when the value in GP timer matches the value in the period register. The value set in **Waveform period** above determines the value in the register.

Note If you select `Period interrupt` and specify a sampling time less than the specified **(Waveform period)/(Event timer clock speed)**, zero-order hold interpolation will occur. (For example, if you enter 64000 as the waveform period, the period for the timer is 64000/75 MHz = 8.5333e-004. If you enter a **Sample time** in the C281x ADC dialog box that is less than this result, it will cause zero-order hold interpolation.)

- `Compare interrupt` — The EV generates an SOC signal for the ADC associated with the same EV module when the value in the GP timer matches the value in the compare register. The value set in **Duty cycle** above determines the value in the register.

See Also

C281x ADC

Purpose

Quadrature encoder pulse circuit

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description

QEP

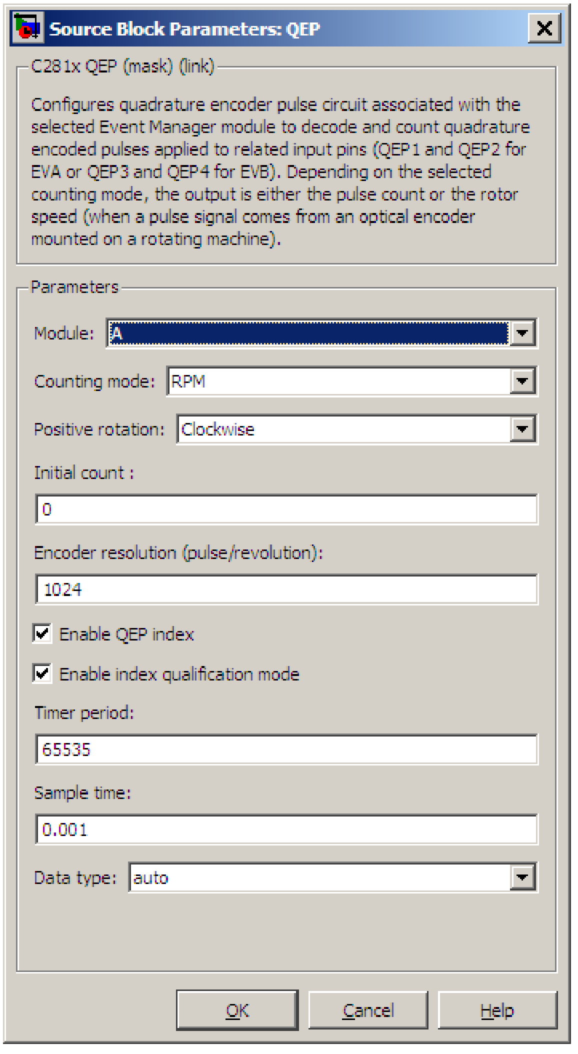
Each F2812 Event Manager has three capture units, which can log transitions on its capture unit pins. Event Manager A (EVA) uses capture units 1, 2, and 3. Event Manager B (EVB) uses capture units 4, 5, and 6.

The quadrature encoder pulse (QEP) circuit decodes and counts quadrature encoded input pulses on these capture unit pins. QEP pulses are two sequences of pulses with varying frequency and a fixed phase shift of 90 degrees (or one-quarter of a period). The circuit counts both edges of the QEP pulses, so the frequency of the QEP clock is four times the input sequence frequency.

The QEP, in combination with an optical encoder, is useful for obtaining speed and position information from a rotating machine. Logic in the QEP circuit determines the direction of rotation by which sequence is leading. For module A, if the QEP1 sequence leads, the general-purpose (GP) Timer counts up and if the QEP2 sequence leads, the timer counts down. The pulse count and frequency determine the angular position and speed.

The C281x QEP module shares GP Timers with other C281 blocks. For more information and guidance on sharing timers, see “Sharing General Purpose Timers between C281x Peripherals”.

Dialog Box



Module

Specify which QEP pins to use:

- A — Uses QEP1 and QEP2 pins.

- B — Uses QEP3 and QEP4 pins.

Counting mode

Specify how to count the QEP pulses:

- Counter — Count the pulses based on GP Timer 2 (or GP Timer 4 for EVB).
- RPM — Count the rotations per minute.

Positive rotation

Defines whether to use Clockwise or Counterclockwise as the direction to use as positive rotation. This field appears only if you select RPM.

Initial count

Initial value for the counter. The value defaults to 0.

Encoder resolution (pulse/revolution)

Number of QEP pulses per revolution. This field appears only if you select RPM.

Enable QEP index

Reset the QEP counter to zero when the QEP index input on CAP3_QEPI1 transitions from low to high.

Enable index qualification mode

Qualify the QEP index input on CAP3_QEPI1. Check that the levels on CAP1_QEP1 and CAP2_QEP2 are high before asserting the index signal as valid.

Timer period

Set the length of the timer period in clock cycles. Enter a value from 0 to 65535. The value defaults to 65535.

If you know the length of a clock cycle, you can easily calculate how many clock cycles to set for the timer period. The following calculation determines the length of one clock cycle:

$$Sysclk(150MHz) \rightarrow HISPCLK(1/2) \rightarrow InputClock\Pr escaler(1/128)$$

In this calculation, you divide the System clock frequency of 150 MHz by the high-speed clock prescaler of 2. Then, you divide the resulting value by the timer control input clock prescaler, 128. The resulting frequency is 0.586 MHz. Thus, one clock cycle is 1/.586 MHz, which is 1.706 μs.

Sample time

Time interval, in seconds, between consecutive reads from the QEP pins.

Data type

Data type of the QEP pin data. The circuit reads the data as 16-bit data and then casts it to the selected data type. Valid data types are `auto`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` or `boolean`.

References

For more information on the QEP module, consult the following documents, available at the Texas Instruments Web site:

- *TMS320x280x, 2801x, 2804x Enhanced Quadrature Encoder Pulse (eQEP) Module Reference Guide*, Literature Number SPRU790
- *Using the Enhanced Quadrature Encoder Pulse (eQEP) Module in TMS320x280x, 28xxx as a Dedicated Capture Application Report*, Literature Number SPRAAH1

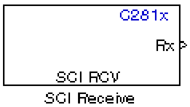
Purpose

Receive data on target via serial communications interface (SCI) from host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description

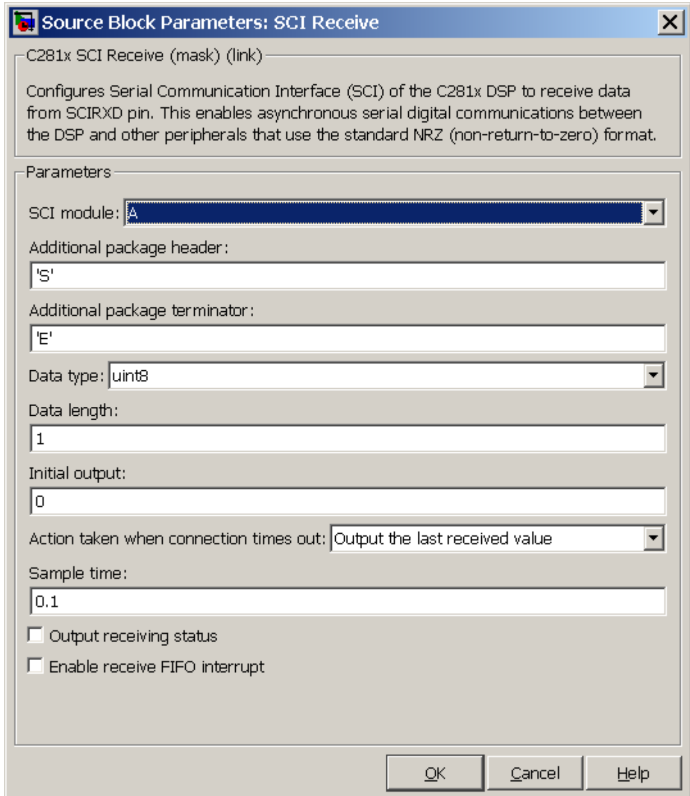


The C281x SCI Receive block supports asynchronous serial digital communications between the target and other asynchronous peripherals in nonreturn-to-zero (NRZ) format. This block configures the C281x DSP target to receive scalar or vector data from the COM port via the C28x target's COM port.

Note For any given model, you can have only one C281x SCI Receive block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

Many SCI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Dialog Box



SCI module

SCI module to be used for communications.

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 may use any 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 Any additional packager header or terminator must match the additional package header or terminator specified in the host 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 may use any 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 Any additional packager header or terminator must match the additional package header or terminator specified in the host SCI Transmit block.

Data type

Data type of the output data. Available options are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

Data length

How many of **Data type** the block will receive (not bytes). Anything more than 1 is a vector. The data length is inherited from the input (the data length originally input to the host-side SCI Transmit block).

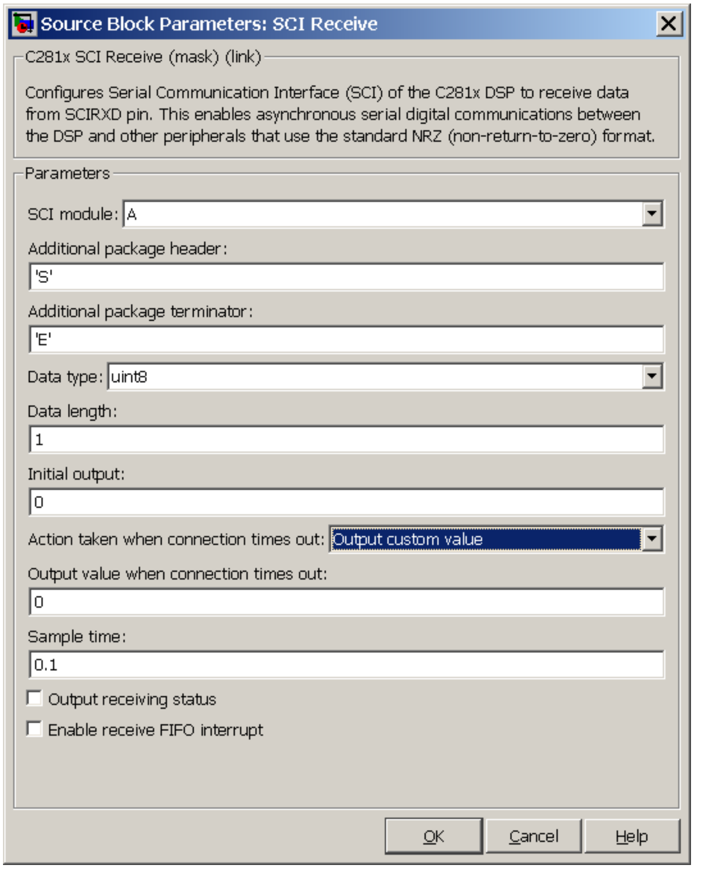
Initial output

Default value from the C281x 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 timeout

Specify what to output if a connection time-out occurs. If “Output the last received value” is selected, the last received value is what is output, unless none has been received yet, in which case the **Initial output** is considered the last received value.

If you select "Output custom value", use the "Output value when connection times out" field to set the custom value.

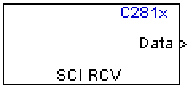


Sample time

Sample time, T_s, for the block's input sampling. To execute this block asynchronously, set **Sample Time** to -1, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

Output receiving status

When this field is checked, the C281x SCI Receive block adds another output port for the transaction status, and appears as shown in the following figure.



SCI Receive

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 between the number of 1s and its parity bit
- 4: SCI framing-error flag — Occurs when an expected stop bit is not found

Enable receive FIFO interrupt

If this option is selected, an interrupt is posted when FIFO is full, allowing the subsystem to take some sort of action (for example, read data as soon as it is received). If this option is cleared, the block stays in polling mode. If the block is in polling mode and not blocking, it checks the FIFO to see if there is data to read. If data is present, it reads and outputs. If no data is present, it continues. If the block is in polling mode and blocking, it waits until data is available to read (when data length is reached).

Receive FIFO interrupt level

This parameter is enabled when the **Enable receive FIFO interrupt** option is selected. Select an interrupt level from 0 to 16. The default level is 0.

References

For detailed information on the SCI module, see *TMS320x281x, 280x DSP Serial Communication Interface (SCI) Reference Guide*, Literature Number SPRU051B, available at the Texas Instruments Web site.

See Also

C281x SCI Transmit

C281x Hardware Interrupt

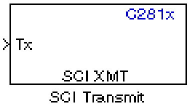
"SCI_A, SCI_B, SCI_C" on page 5-900

Purpose

Transmit data from target via serial communications interface (SCI) to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x



Description

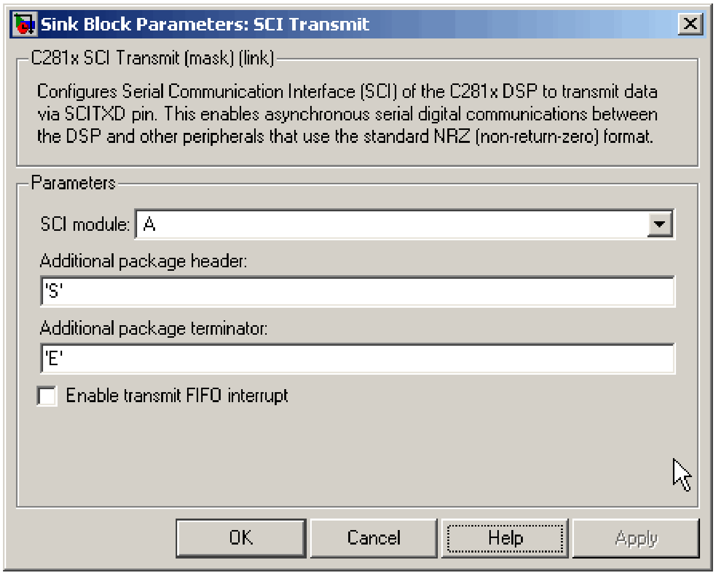
The C281x SCI Transmit block transmits scalar or vector data in `int8` or `uint8` format from the C281x target's COM ports in nonreturn-to-zero (NRZ) format. You can specify how many of the six target COM ports to use. The sampling rate and data type are inherited from the input port. The data type of the input port must be one of the following: single, int8, uint8, int16, uint16, int32, or uint32. If no data type is specified, the default data type is `uint8`.

Note For any given model, you can have only one C281x SCI Transmit block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

Many SCI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Fixed-point inputs are not supported for this block.

Dialog Box



SCI module

SCI module to be used for communications.

Additional package header

This field specifies the data located at the front of the sent 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 may use any 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 Any additional packager header or terminator must match the additional package header or terminator specified in the host SCI Receive block.

Additional package terminator

This field specifies the data located at the end of the sent data package, which is not part of the data being transmitted, and generally indicates end of data. The additional package terminator must be an ASCII value. You may use any 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 Any additional packager header or terminator must match the additional package header or terminator specified in the host SCI Receive block.

Enable transmit FIFO interrupt

If this option is selected, an interrupt is posted when FIFO is full, allowing the subsystem to take some sort of action.

References

For detailed information on the SCI module, see *TMS320x281x, 280x DSP Serial Communication Interface (SCI) Reference Guide*, Literature Number SPRU051B, available at the Texas Instruments Web site.

See Also

C281x SCI Receive

C281x Hardware Interrupt

"SCI_A, SCI_B, SCI_C" on page 5-900

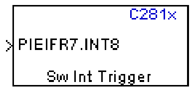
C281x Software Interrupt Trigger

Purpose

Generate software triggered nonmaskable interrupt

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x



Software Interrupt Trigger

Description

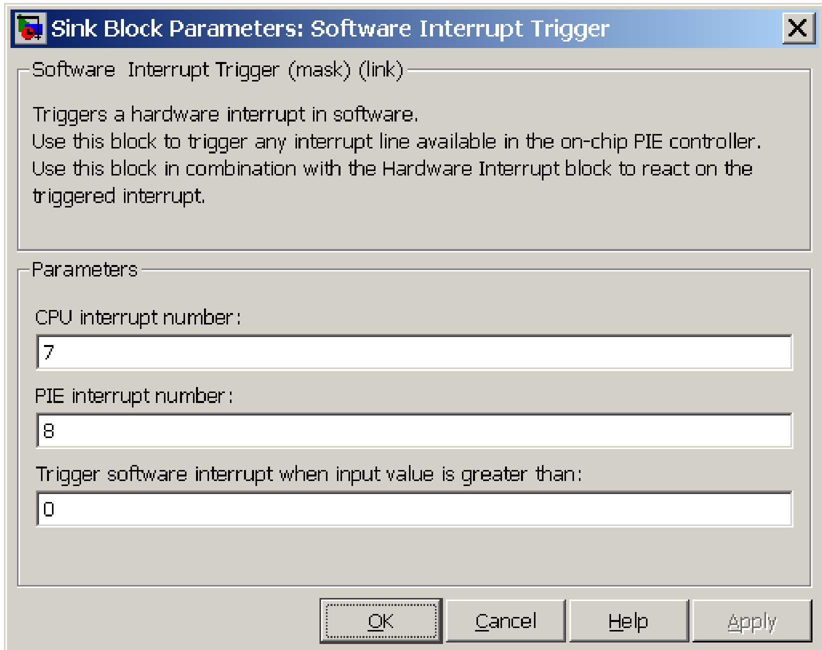
When you add this block to a model, the block polls the input port for the input value. When the input value is greater than the value in **Trigger software interrupt when input value is greater than**, the block posts the interrupt to a Hardware Interrupt block in the model.

To use this block, add a Hardware Interrupt block to your model to process the software triggered interrupt from this block into an interrupt service routine on the processor. Set the interrupt number in the Hardware Interrupt block to the value you set here in **CPU interrupt number**.

The CPU and PIE interrupt numbers together specify a single interrupt for a single peripheral or peripheral module. The C281x Peripheral Interrupt Vector Values on page 5-131 table maps CPU and PIE interrupt numbers to these peripheral interrupts.

Note Fixed-point inputs are not supported for this block.

Dialog Box



CPU interrupt number
: Specify the interrupt the block responds to. Interrupt numbers are integers ranging from 1 to 12.

PIE interrupt number
: Enter an integer value from 1 to 8 to set the Peripheral Interrupt Expansion (PIE) interrupt number.

Trigger software interrupt when input value is greater than:
: Sets the value above which the block posts an interrupt. Enter the value to set the level that indicates that the interrupt is asserted by a requesting routine.

References

For detailed information about interrupt processing, see *TMS320x281x DSP System Control and Interrupts Reference Guide*, SPRU078C, available at the Texas Instruments Web site.

See Also

C281x Hardware Interrupt

C281x SPI Receive

Purpose

Receive data via serial peripheral interface on target

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description

SPI Receive

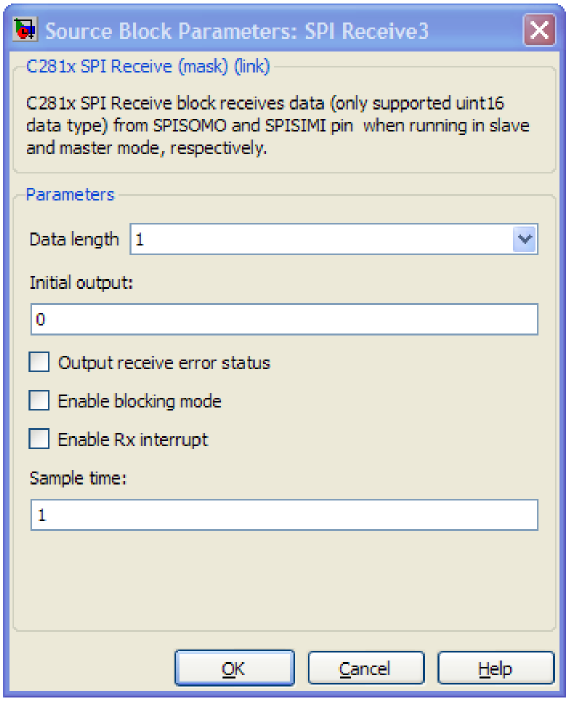
The C281x SPI Receive supports synchronous, serial peripheral input/output port communications between the DSP controller and external peripherals or other controllers. The block can run in either slave or master mode.

In master mode, the SPISIMO pin transmits data and the SPISOMI pin receives data. When master mode is selected, the SPI initiates the data transfer by sending a serial clock signal (SPICLK), which is used for the entire serial communications link. Data transfers are synchronized to this SPICLK, which enables both master and slave to send and receive data simultaneously. The maximum for the clock is one quarter of the DSP controller's clock frequency.

For any given model, you can have only one C281x SPI Receive block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

Note Many SPI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Dialog Box



Data length

Specify how many `uint16`s are expected to be received. Select 1 through 16.

Initial output

Set the value the SPI node outputs to the model before it has received any data.

The default value is `0`.

Enable blocking mode

If this option is selected, system waits until data is received before continuing processing.

Output receive error status

When this field is checked, the C281x SPI Receive block adds another output port for the transaction status, and appears as shown in the following figure.

Error status may be one of the following values:

- `0`: No errors
- `1`: Data loss occurred (Overrun: when FIFO disabled, Overflow: when FIFO enabled)
- `2`: Data not ready, a time-out occurred while the block was waiting to receive data

Post interrupt when data is received

Check this check box to post an asynchronous interrupt when data is received.

Sample time

Sample time, T_s, for the block's input sampling. To execute this block asynchronously, set **Sample Time** to `-1`, check the **Post interrupt when message is received** box, and refer to "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

See Also

C281x SPI Transmit

C281x Hardware Interrupt

"SPI_A, SPI_B, SPI_C, SPI_D" on page 5-904

C281x SPI Transmit

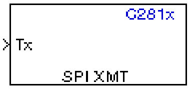
Purpose

Transmit data via serial peripheral interface (SPI) to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description



SPI Transmit

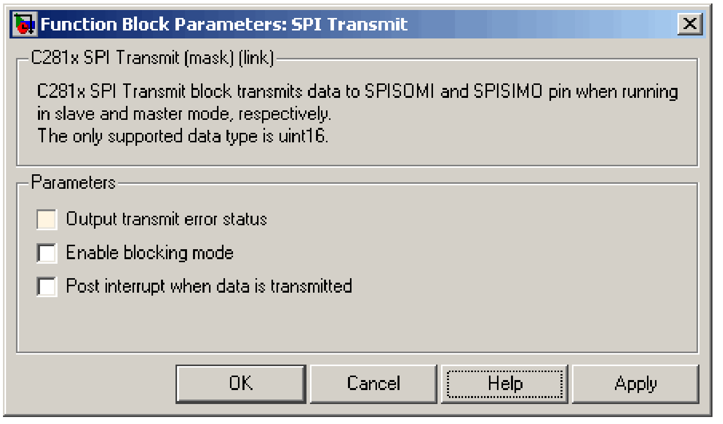
The C281x SPI Transmit supports synchronous, serial peripheral input/output port communications between the DSP controller and external peripherals or other controllers. The block can run in either slave or master mode. In master mode, the SPISIMO pin transmits data and the SPISOMI pin receives data. When master mode is selected, the SPI initiates the data transfer by sending a serial clock signal (SPICLK), which is used for the entire serial communications link. Data transfers are synchronized to this SPICLK, which enables both master and slave to send and receive data simultaneously. The maximum for the clock is one quarter of the DSP controller's clock frequency.

The sampling rate is inherited from the input port. The supported data type is `uint16`.

Note For any given model, you can have only one C281x SPI Transmit block per module. There are two modules, A and B, which can be configured through the Target Preferences block.

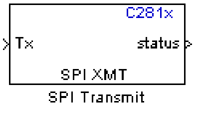
Many SPI-specific settings are in the **DSPBoard** section of the Target Preferences block. You should verify that these settings are correct for your application.

Dialog Box



Output transmit error status

When this field is checked, the C281x SPI Transmit block adds another output port for the transaction status, and appears as shown in the following figure.



Error status may be one of the following values:

- `0`: No errors
- `1`: A time-out occurred while the block was transmitting data
- `2`: There is an error in the transmitted data (for example, header or terminator don't match, length of data expected is too big or too small)

Enable blocking mode

If this option is selected, system waits until data is sent before continuing processing.

Post interrupt when data is transmitted
Select this check box to post an asynchronous interrupt when data is transmitted.

See Also

C281x SPI Receive

"SPI_A, SPI_B, SPI_C, SPI_D" on page 5-904

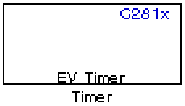
Purpose

Configure general-purpose timer in Event Manager module

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Description

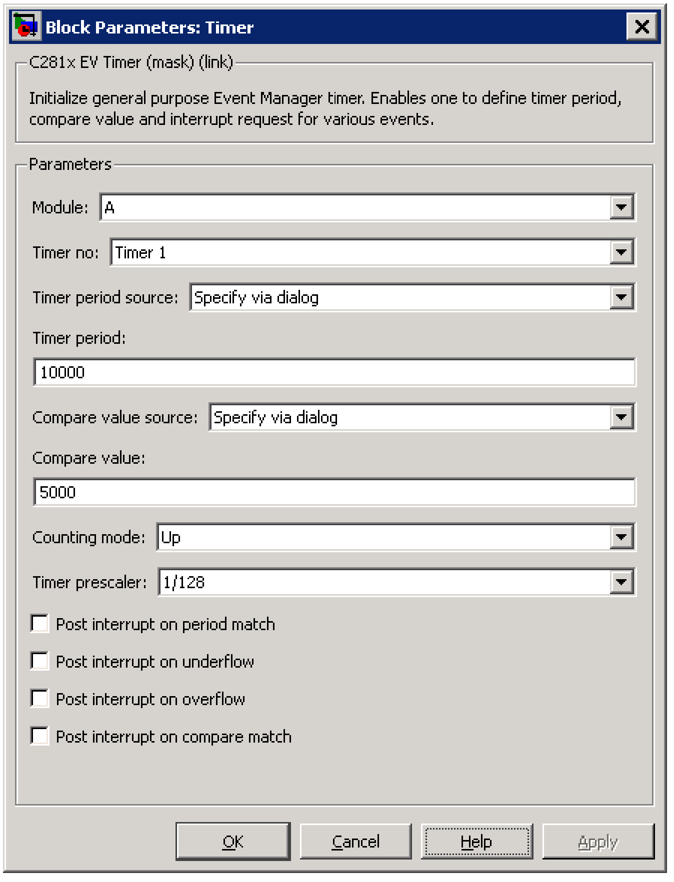


The C281x contains two event-manager (EV) modules. Each module contains two general-purpose (GP) timers. You can use these timers as independent time bases for various applications.

Use the C281x Timer block to set the periodicity of one GP timer and the conditions under which it posts interrupts. Each model can contain up to four C281x Timer blocks.

The C281x Timer module configures GP Timers that other C281 blocks share. For more information and guidance on sharing timers, see "Sharing General Purpose Timers between C281x Peripherals".

Dialog Box



Module
Timer no

Select which of four possible timers to configure. Setting **Module** to `A` lets you select `Timer 1` or `Timer 2` in **Timer no**. Setting **Module** to `B` lets you select `Timer 3` or `Timer 4` in **Timer no**.

Clock source

When **Timer no** has a value of `Timer 2` or `Timer 4`, use this parameter to select the clock source for the event timer. You can choose either `Internal` or `QEP circuit`. When you select `Internal`, you can configure other options such as **Timer period source**, **Counting mode**, and **Timer prescaler**.

Timer period source

Select the source of the event timer period. Use `Specify via dialog` to set the period using **Timer period**. Select `Input port` to create an input, **T**, that accepts the value of the timer period in clock cycles, from `0` to `65535`. **Timer period source** becomes unavailable when **Clock source** is set to `QEP circuit`.

Timer period

Set the length of the timer period in clock cycles. Enter a value from `0` to `65535`. The value defaults to `10000`.

If you know the length of a clock cycle, you can easily calculate how many clock cycles to set for the timer period. The following calculation determines the length of one clock cycle:

$$Sysclk(150MHz) \rightarrow HISPCLK(1/2) \rightarrow InputClock\,\mathrm{Pr}\,escaler(1/128)$$

In this calculation, you divide the System clock frequency of 150 MHz by the high-speed clock prescaler of 2. Then, you divide the resulting value by the timer control input clock prescaler, 128. The resulting frequency is 0.586 MHz. Thus, one clock cycle is 1/.586 MHz, which is 1.706 μs.

Compare value source

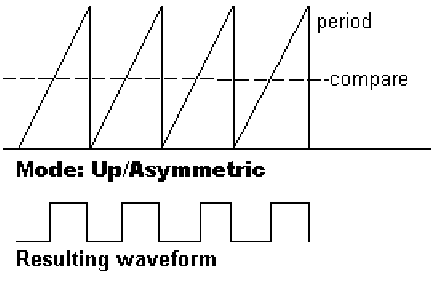
Select the source of the compare value. Use `Specify via dialog` to set the period using the **Compare value** parameter. Select `Input port` to create a block input, **W**, that accepts the value of the compare value, from `0` to `65535`.

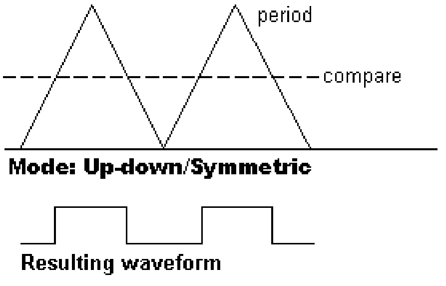
Compare value

Enter a constant value for comparison to the running timer value for generating interrupts. Enter a value from `0` to `65535`. The value defaults to `5000`. The timer only generates interrupts if you enable **Post interrupt on compare match**.

Counting mode

Select `Up` to generate an asymmetrical waveform output, or `Up-down` to generate a symmetrical waveform output, as shown in the following illustration.





When you specify the **Counting mode** as `Up` (asymmetric) the waveform:

- Starts low
- Goes high when the rising period counter value matches the **Compare value**
- Goes low at the end of the period

When you specify the **Counting mode** as `Up-down` (symmetric) the waveform:

- Starts low
- Goes high when the increasing period counter value matches the **Compare value**
- Goes low when the decreasing period counter value matches the **Compare value**

Counting mode becomes unavailable when **Clock source** is set to `QEP circuit`.

Timer prescaler
Divide the clock input to produce the desired timer counting rate.

Timer prescaler becomes unavailable when **Clock source** is set to `QEP circuit`.

Post interrupt on period match
Generate an interrupt when the value of the timer reaches its maximum value as specified in **Timer period**.

Post interrupt on underflow
Generate an interrupt when the value of the timer cycles back to `0`.

Post interrupt on overflow
Generate an interrupt when the value of the timer reaches its maximum, `65535`. Also set **Timer period** to `65535` for this parameter to work.

Post interrupt on compare match
Generate an interrupt when the value of the timer equals **Compare value**.

References

TMS320x281x DSP Event Manager (EV) Reference Guide, Literature Number: SPRU065, available from the Texas Instruments Web site.

See Also

C281x Hardware Interrupt, Idle Task

C28x Watchdog

Purpose

Configure counter reset source of DSP Watchdog module

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C280x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2802x

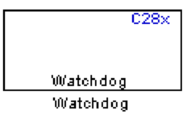
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2803x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C281x

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C28x3x

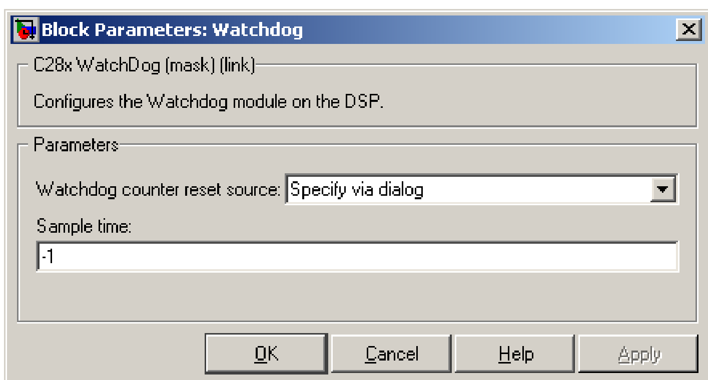
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ C2834x

Description



This block configures the counter reset source of the Watchdog module on the DSP.

Dialog Box



Watchdog counter reset source

- `Input` — Create a input port on the watchdog block. The input signal resets the counter.
- `Specify via dialog` — Use the value of **Sample time** to reset the watchdog timer.

Sample time

The interval at which the DSP resets the watchdog timer. 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 "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

See Also

"Watchdog" on page 5-909

C2000 Clarke Transformation

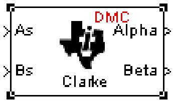
Purpose

Convert balanced three-phase quantities to balanced two-phase quadrature quantities

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description

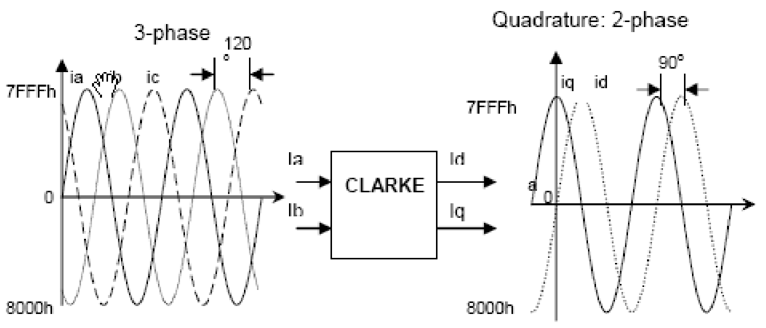


Clarke Transformation

This block converts balanced three-phase quantities into balanced two-phase quadrature quantities. The transformation implements these equations

$$Id = Ia$$
$$Iq = (2Ib + Ia) / \sqrt{3}$$

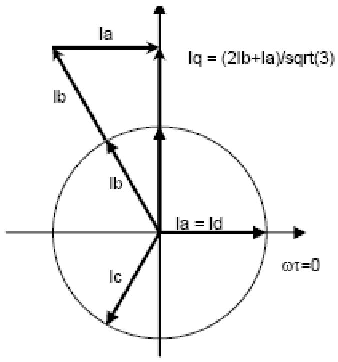
and is illustrated in the following figure.



The inputs to this block are the phase a (`As`) and phase b (`Bs`) components of the balanced three-phase quantities and the outputs are the direct axis (`Alpha`) component and the quadrature axis (`Beta`) of the transformed signal.

The instantaneous outputs are defined by the following equations and are shown in the following figure:

$$ia = I * \sin(\omega t)$$
$$ib = I * \sin(\omega t + 2\pi / 3)$$
$$ic = I * \sin(\omega t - 2\pi / 3)$$
$$id = I * \sin(\omega t)$$
$$iq = I * \sin(\omega t + \pi / 2)$$



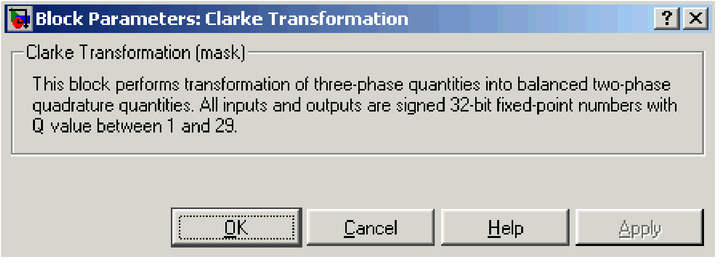
The variables used in the preceding equations and figures correspond to the variables on the block as shown in the following table:

	Equation Variables	Block Variables
Inputs	ia	As
	ib	Bs
Outputs	id	Alpha
	iq	Beta

Note

- To generate optimized code from this block, enable the `TI C28x` or `TI C28x (ISO)` TFL. See“About Target Function Libraries and Optimization for Embedded TargetsDesktop Targets”.
- The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See “Using the IQmath Library” for more information.

Dialog Box



References

For detailed information on the DMC library, see *C/F 28xx Digital Motor Control Library*, Literature Number SPRC080, available at the Texas Instruments Web site.

See Also

C2000 Inverse Park Transformation, C2000 Park Transformation, C2000 PID Controller, C2000 Space Vector Generator, C2000 Speed Measurement

C2000 Division IQN

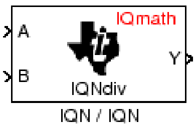
Purpose

Divide IQ numbers

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

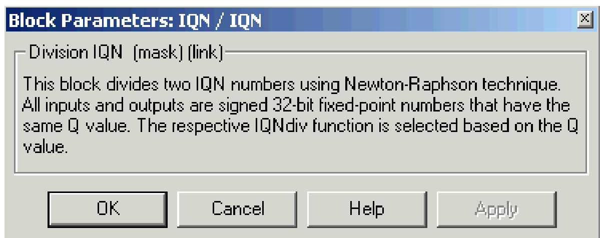
Description



This block divides two numbers that use the same Q format, using the Newton-Raphson technique. The resulting quotient uses the same Q format at the inputs.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, c2000 Arctangent IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000

IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

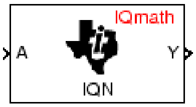
Purpose

Convert floating-point number to IQ number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description

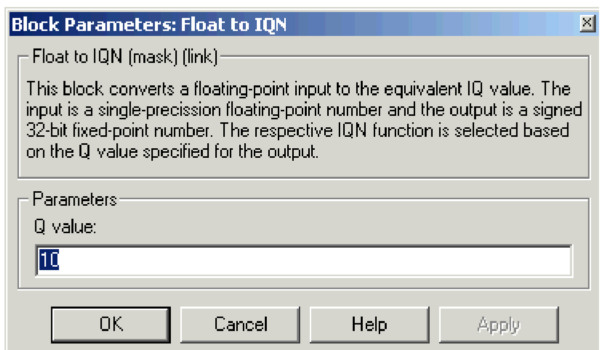


Float to IQN

This block converts a floating-point number to an IQ number. The Q value of the output is specified in the dialog.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Q value

Q value from 1 to 30 that specifies the precision of the output

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

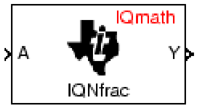
Purpose

Fractional part of IQ number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

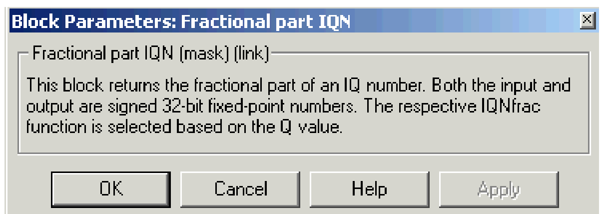


Fractional part IQN

This block returns the fractional portion of an IQ number. The returned value is an IQ number in the same IQ format.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

C2000 Fractional part IQN x int32

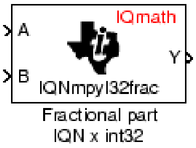
Purpose

Fractional part of result of multiplying IQ number and long integer

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

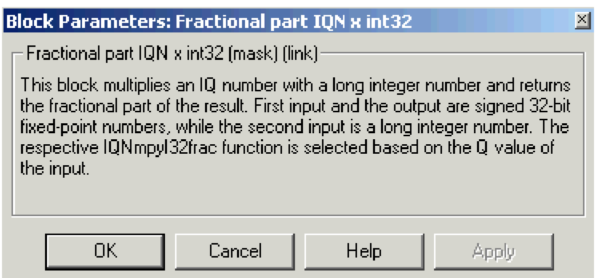
Description



This block multiplies an IQ input and a long integer input and returns the fractional portion of the resulting IQ number.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

C2000 From RTDX

Purpose

Add RTDX communication channel for target to receive data from host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ RTDX Instrumentation

Description

From RTDX

Note This block will be removed from the Embedded Coder product in an upcoming release.

Note To use RTDX for C28x host/target communications, download and install TI DSP/BIOS. The DSP/BIOS installation includes files required for RTDX communications. For more information, see *DSP/BIOS, RTDX and Host-Target Communications*, Literature Number SPRA895, available at the Texas Instruments Web site.

When you generate code from Simulink in Simulink Coder software with a From RTDX block in your model, code generation inserts the C commands to create an RTDX input channel on the target. Input channels transfer data from the host to the target.

The generated code contains this command:

```
RTDX_enableInput(&channelname)
```

where `channelname` is the name you enter in **Channel name.**

Note From RTDX blocks work only in code generation and when your model runs on your target. In simulations, this block does not perform any operations, except generating an output matching your specified initial conditions.

To use RTDX blocks in your model, you must do the following:

1 Add one or more To RTDX or From RTDX blocks to your model.

2 Download and run your model on your target.

3 Enable the RTDX channels from MATLAB or use **Enable RTDX channel on start-up** on the block dialog.

4 Use the `readmsg` and `writemsg` functions on the MATLAB command line to send and retrieve data from the target over RTDX.

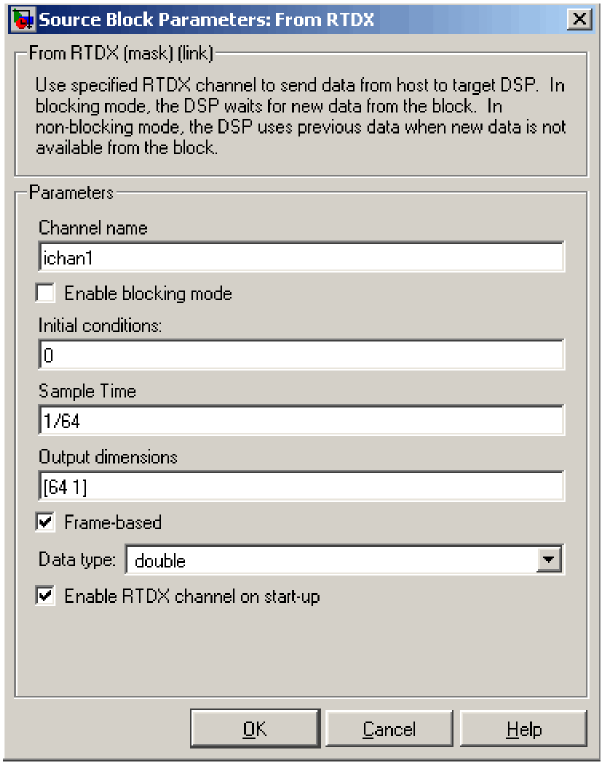
,For more information about using RTDX in your model, see the following demos:

- Real-Time Data Exchange (RTDX™) Tutorial
- Comparing Simulation and Target Implementation with RTDX
- Real-Time Data Exchange via RTDX
- DC Motor Speed Control via RTDX™

Note To use RTDX with the XDS100 USB JTAG Emulator and the C28027 chip, add the following line to the linker command file:

```
_RTDX_interrupt_mask = ~0x000000008;
```

Dialog Box



Channel name

Name of the input channel to be created by the generated code. The channel name must meet C syntax requirements for length and character content.

Enable blocking mode

Blocking mode instructs the target processor to pause processing until new data is available from the From RTDX block. If you enable blocking and new data is not available when the processor needs it, your process stops. In nonblocking mode, the processor uses old data from the block when new data is not available.

Nonblocking operation is the default and is recommended for most operations.

Initial conditions

Data the processor reads from RTDX for the first read. If blocking mode is not enabled, you must have an entry for this option. Leaving the option blank causes an error in Simulink Coder software. Valid values are 0, null ([ ]), or a scalar. The default value is 0.

0 or null ([ ]) outputs a zero to the processor. A scalar generates one output sample with the value of the scalar. If **Output dimensions** specifies an array, every element in the array has the same scalar or zero value. A null array ([ ]) outputs a zero for every sample.

Sample time

Time between samples of the signal. The value defaults to 1 second. This produces a sample rate of one sample per second (1/**Sample time**).

Output dimensions

Dimensions of a matrix for the output signal from the block. The first value is the number of rows and the second is the number of columns. For example, the default setting `[1 64]` represents a 1-by-64 matrix of output values. Enter a 1-by-2 vector for the dimensions.

Frame-based

Sets a flag at the block output that directs downstream blocks to use frame-based processing on the data from this block. In frame-based processing, the samples in a frame are processed simultaneously. In sample-based processing, samples are processed one at a time. Frame-based processing can increase the speed of your application running on your target. Throughput remains the same in samples per second processed. Frame-based operation is the default.

Data type

Type of data coming from the block. Select one of the following types:

- `Double` — Double-precision floating-point values. This is the default. Values range from -1 to 1.
- `Single` — Single-precision floating-point values ranging from -1 to 1.
- `Uint8` — 8-bit unsigned integers. Output values range from 0 to 255.
- `Int16` — 16-bit signed integers. With the sign, the values range from -32768 to 32767.
- `Int32` — 32-bit signed integers. Values range from -2^{31} to $(2^{31}-1)$.

Enable RTDX channel on start-up

Enables the RTDX channel when you start the channel from MATLAB. With this selected, you do not need to use the `enable` function to prepare your RTDX channels. This option applies only to the channel you specify in **Channel name**. You do have to open the channel.

See Also

`ticcs`, `readmsg`, C2000 To RTDX, `writemsg`.

References

RTDX 2.0 User's Guide, Literature Number: SPRUFC7, available from the Texas Instruments Web site.

How to Write an RTDX Host Application Using MATLAB, Literature Number: SPRA386, available from the Texas Instruments Web site.

C2000 Integer part IQN

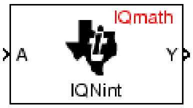
Purpose

Integer part of IQ number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

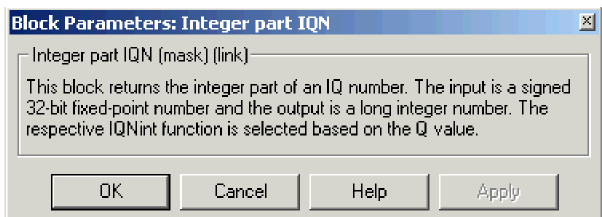


Integer part IQN

This block returns the integer portion of an IQ number. The returned value is a long integer.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

C2000 Integer part IQN x int32

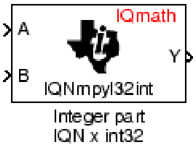
Purpose

Integer part of result of multiplying IQ number and long integer

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

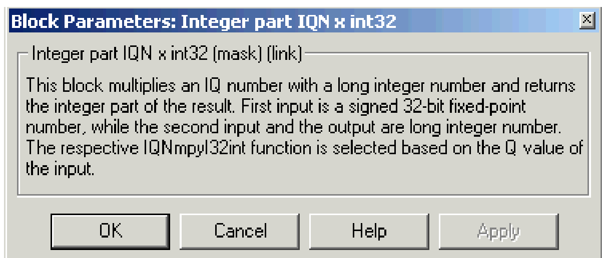
Description



This block multiplies an IQ input and a long integer input and returns the integer portion of the resulting IQ number as a long integer.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

C2000 Inverse Park Transformation

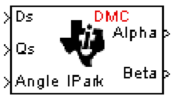
Purpose

Convert rotating reference frame vectors to two-phase stationary reference frame

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

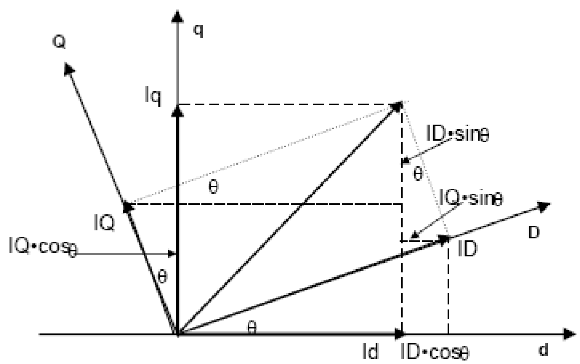
Description



This block converts vectors in an orthogonal rotating reference frame to a two-phase orthogonal stationary reference frame. The transformation implements these equations:

$$Id = ID * \cos\theta - IQ * \sin\theta$$
$$Iq = ID * \sin\theta + IQ * \cos\theta$$

and is illustrated in the following figure.



The inputs to this block are the direct axis (`Ds`) and quadrature axis (`Qs`) components of the transformed signal in the rotating frame and the phase angle (`Angle`) between the stationary and rotating frames.

The outputs are the direct axis (`Alpha`) and the quadrature axis (`Beta`) components of the transformed signal.

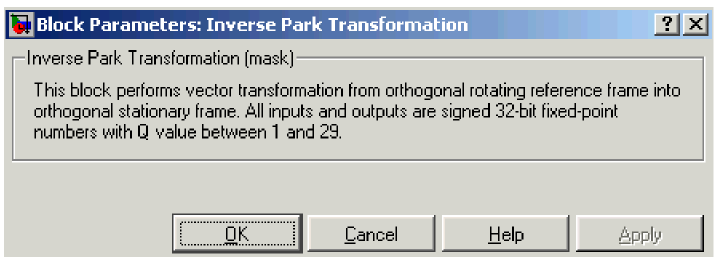
The variables used in the preceding figure and equations correspond to the block variables as shown in the following table:

	Equation Variables	Block Variables
Inputs	ID	Ds
	IQ	Qs
	θ	Angle
Outputs	id	Alpha
	iq	Beta

Note

- To generate optimized code from this block, enable the `TI C28x` or `TI C28x (ISO)` TFL. See“About Target Function Libraries and Optimization for Embedded TargetsDesktop Targets”.
- The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See “Using the IQmath Library” for more information.

Dialog Box



References

For detailed information on the DMC library, see *C/F 28xx Digital Motor Control Library*, Literature Number SPRC080, available at the Texas Instruments Web site.

See Also

C2000 Clarke Transformation, C2000 Park Transformation, C2000 PID Controller, C2000 Space Vector Generator, C2000 Speed Measurement

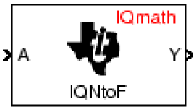
Purpose

Convert IQ number to floating-point number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

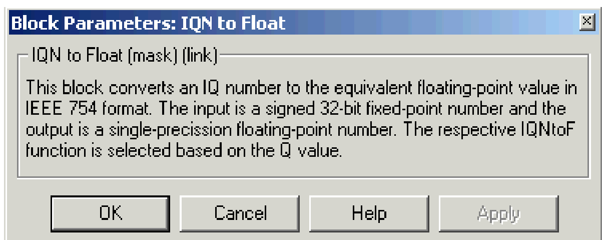


IQN to Float

This block converts an IQ input to an equivalent floating-point number. The output is a single floating-point number.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000

IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

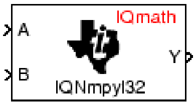
Purpose

Multiply IQ number with long integer

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

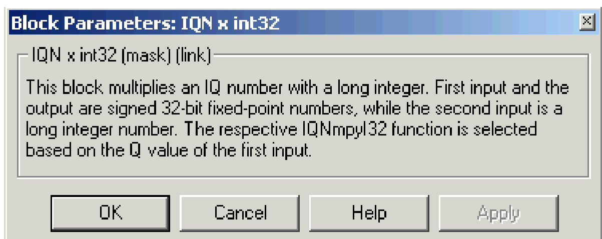


IQN x int32

This block multiplies an IQ input and a long integer input and produces an IQ output of the same Q value as the IQ input.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000

IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

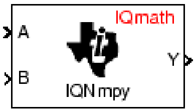
Purpose

Multiply IQ numbers with same Q format

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

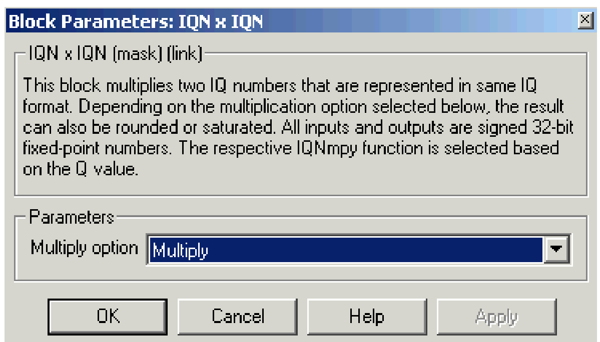


IQN x IQN

This block multiplies two IQ numbers. Optionally, it can also round and saturate the result.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Multiply option

Type of multiplication to perform:

- `Multiply` — Multiply the numbers.
- `Multiply with Rounding` — Multiply the numbers and round the result.

- `Multiply with Rounding and Saturation` — Multiply the numbers and round and saturate the result to the maximum value.

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

Purpose

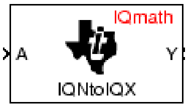
Convert IQ number to different Q format

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

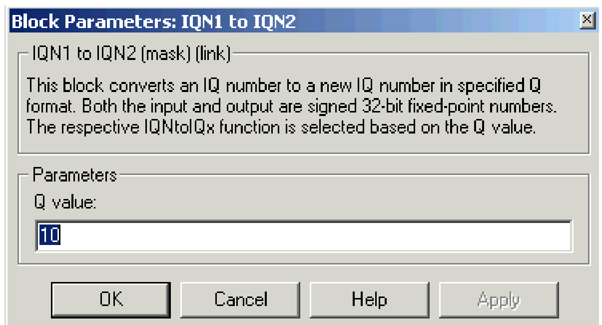
This block converts an IQ number in a particular Q format to a different Q format.



IQN1 to IQN2

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Q value

Q value from 1 to 30 that specifies the precision of the output

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

C2000 IQN1 x IQN2

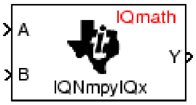
Purpose

Multiply IQ numbers with different Q formats

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

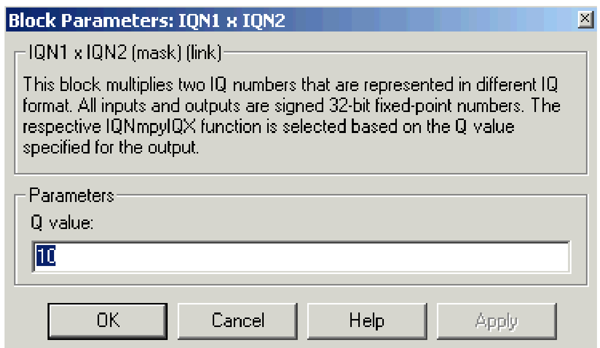


IQN1 x IQN2

This block multiplies two IQ numbers when the numbers are represented in different Q formats. The format of the result is specified in the dialog box.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Q value

Q value from 1 to 30 that specifies the precision of the output

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

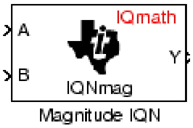
Purpose

Magnitude of two orthogonal IQ numbers

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description



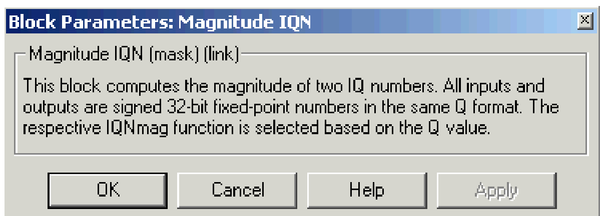
This block calculates the magnitude of two IQ numbers using

$$\sqrt{a^2 + b^2}$$

The output is an IQ number in the same Q format as the input.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Saturate IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

C2000 Park Transformation

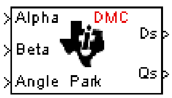
Purpose

Convert two-phase stationary system vectors to rotating system vectors

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description

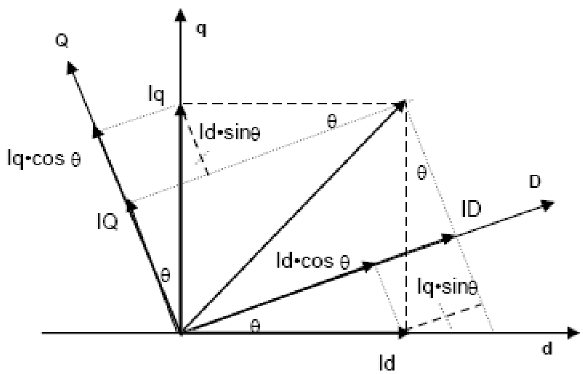


Park Transformation

This block converts vectors in balanced two-phase orthogonal stationary systems into an orthogonal rotating reference frame. The transformation implements these equations

$$ID = Id * \cos\theta + Iq * \sin\theta$$
$$IQ = -Id * \sin\theta + Iq * \cos\theta$$

and is illustrated in the following figure.



The variables used in the preceding figure and equations correspond to the block variables as shown in the following table:

	Equation Variables	Block Variables
Inputs	id	Alpha
	iq	Beta
	θ	Angle

	Equation Variables	Block Variables
Outputs	ID	Ds
	IQ	Qs

The inputs to this block are the direct axis (Alpha) and the quadrature axis (Beta) components of the transformed signal and the phase angle (Angle) between the stationary and rotating frames.

The outputs are the direct axis (Ds) and quadrature axis (Qs) components of the transformed signal in the rotating frame.

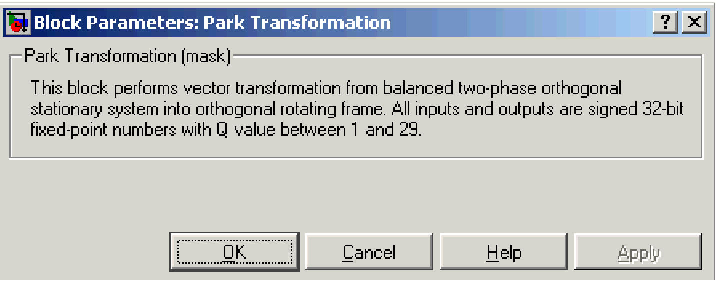
The instantaneous inputs are defined by the following equations:

$$id = I * \sin(\omega t)$$
$$iq = I * \sin(\omega t + \pi / 2)$$

Note

- To generate optimized code from this block, enable the TI C28x or TI C28x (ISO) TFL. See“About Target Function Libraries and Optimization for Embedded TargetsDesktop Targets”.
- The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See “Using the IQmath Library” for more information.

Dialog Box



References

For detailed information on the DMC library, see *C/F 28xx Digital Motor Control Library*, Literature Number SPRC080, available at the Texas Instruments Web site.

See Also

C2000 Clarke Transformation, C2000 Inverse Park Transformation, C2000 PID Controller, C2000 Space Vector Generator, C2000 Speed Measurement

C2000 PID Controller

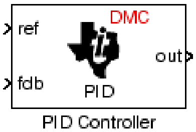
Purpose

Digital PID controller

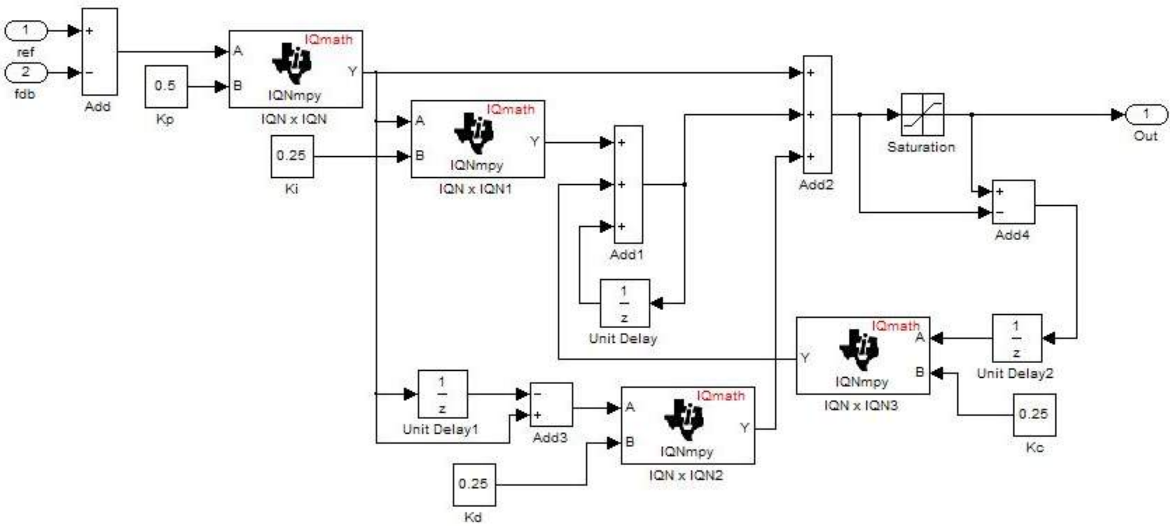
Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description



This block implements a 32-bit digital PID controller with antiwindup correction. The inputs are a reference input (`ref`) and a feedback input (`fdb`) and the output (`out`) is the saturated PID output. The following diagram shows a PID controller with antiwindup.



The differential equation describing the PID controller before saturation that is implemented in this block is

“$u_{presat}(t) = u_p(t) + u_i(t) + u_d(t)$”

where u_{presat} is the PID output before saturation, u_p is the proportional term, u_i is the integral term with saturation correction, and u_d is the derivative term.

The proportional term is

"$u_p(t) = K_p e(t)$"

where K_p is the proportional gain of the PID controller and *e(t)* is the error between the reference and feedback inputs.

The integral term with saturation correction is

$$u_i(t) = \int_0^t \left\{ \frac{K_p}{T_i} e(\tau) + K_c \left(u(\tau) - u_{presat}(\tau) \right) \right\} d\tau$$

where K_c is the integral correction gain of the PID controller.

The derivative term is

$$u_d(t) = K_p T_d \frac{de(t)}{dt}$$

where T_d is the derivative time of the PID controller. In discrete terms, the derivative gain is defined as $K_d = T_d/T$, and the integral gain is defined as $K_i = T/T_i$, where T is the sampling period and T_i is the integral time of the PID controller.

Using backward approximation, the preceding differential equations can be transformed into the following discrete equations.

$$u_p[n] = K_p e[n]$$

$$u_i[n] = u_i[n-1] + K_i K_p e[n] + K_c \left(u[n-1] - u_{presat}[n-1] \right)$$

$$u_d[n] = K_d K_p \left(e[n] - e[n-1] \right)$$

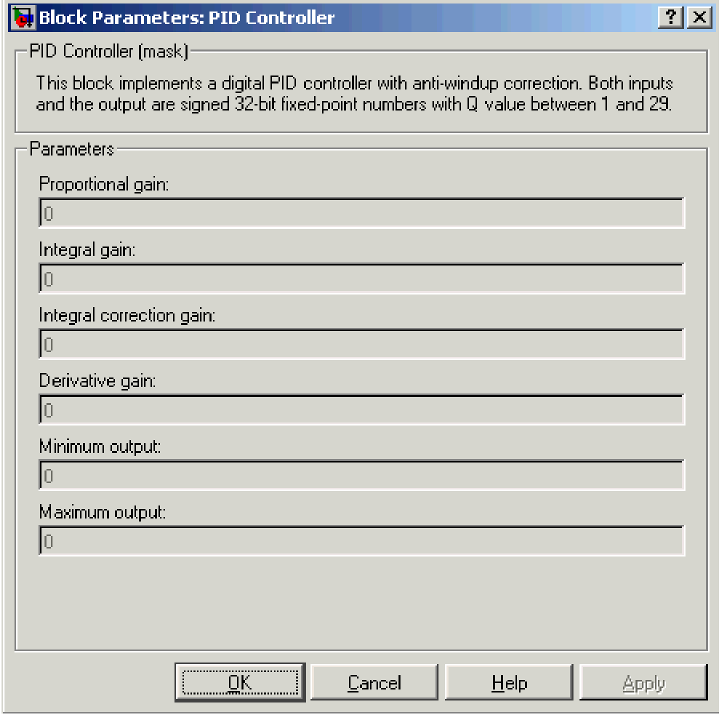
$$u_{presat}[n] = u_p[n] + u_i[n] + u_d[n]$$

$$u[n] = SAT\left(u_{presat}[n] \right)$$

Note

- To generate optimized code from this block, enable the `TI C28x` or `TI C28x (ISO)` TFL. See“About Target Function Libraries and Optimization for Embedded TargetsDesktop Targets”.
- The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See “Using the IQmath Library” for more information.

Dialog Box



Proportional gain
Amount of proportional gain (K_p) to apply to the PID

Integral gain
Amount of gain (K_i) to apply to the integration equation

Integral correction gain
Amount of correction gain (K_c) to apply to the integration equation

Derivative gain
Amount of gain (K_d) to apply to the derivative equation.

Minimum output
Minimum allowable value of the PID output

Maximum output
Maximum allowable value of the PID output

References

For detailed information on the DMC library, see *C/F 28xx Digital Motor Control Library*, Literature Number SPRC080, available at the Texas Instruments Web site.

See Also

C2000 Clarke Transformation, C2000 Inverse Park Transformation, C2000 Park Transformation, C2000 Space Vector Generator, C2000 Speed Measurement

C2000 Ramp Control

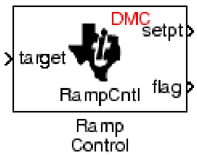
Purpose

Create ramp-up and ramp-down function

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description

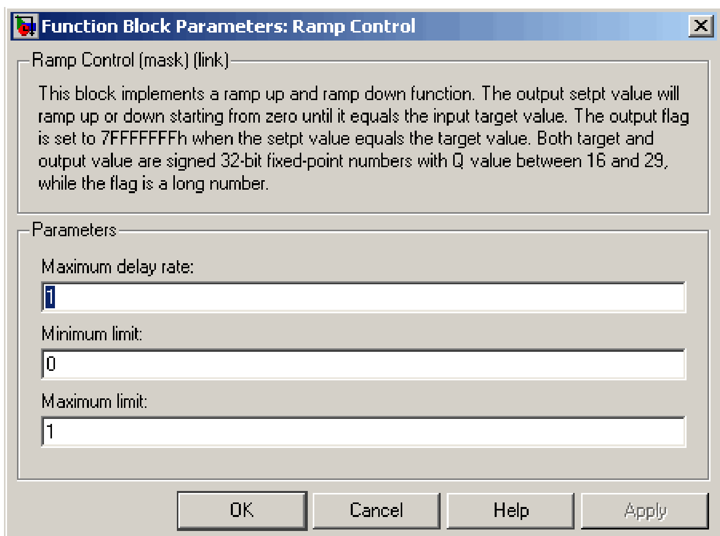


This block implements a ramp-up and ramp-down function. The input is a `target` value and the outputs are the set point value (`setpt`) and a `flag`. The `flag` output is set to `7FFFFFFFh` when the output `setpt` value reaches the input `target` value. The `target` and `setpt` values are signed 32-bit fixed-point numbers with Q values between 16 and 29. The flag is a long number.

The `target` value is compared with the `setpt` value. If they are not equal, the output `setpt` is adjusted up or down by a fixed step size (0.0000305).

If the fixed step size is relatively large compared to the `target` value, the output may oscillate around the `target` value.

Dialog Box



Maximum delay rate
Value that is multiplied by the sampling loop time period to determine the time delay for each ramp step. Valid values are integers greater than 0.

Minimum limit
Minimum allowable ramp value. If the input falls below this value, it will be saturated to this minimum. The smallest value you can enter is the minimum value that can be represented in fixed-point data format by the input and output blocks to which this Ramp Control block is connected in your model. If you enter a value below this minimum, an error occurs at the start of code generation or simulation. For example, if your input is in Q29 format, its minimum value is -4.

Maximum limit
Maximum allowable ramp value. If the input goes above this value, it will be reduced to this maximum. The largest value you can enter is the maximum value that can be represented in fixed-point data format by the input and output blocks to which this Ramp Control block is connected in your model. If you enter a value above this maximum, an error occurs at the start of code generation or simulation. For example, if your input is in Q29 format, its maximum value is 3.9999....

See Also

C2000 Ramp Generator

C2000 Ramp Generator

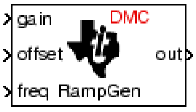
Purpose

Generate ramp output

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description



Ramp Generator

This block generates ramp output (`out`) from the slope of the ramp signal (`gain`), DC offset in the ramp signal (`offset`), and frequency of the ramp signal (`freq`) inputs. All of the inputs and output are 32-bit fixed-point numbers with Q values between 1 and 29.

Algorithm

The block's output (`out`) at the sampling instant k is governed by the following algorithm:

“`out`(k) = angle(k) * `gain`(k) + `offset`(k) ”

For `out`(k) > 1, `out`(k) = `out`(k) - 1. For `out`(k) < -1, `out`(k) = `out`(k) + 1.

Angle(k) is defined as follows:

“angle(k) = angle(k-1) + `freq`(k) * **Maximum step angle**

for angle(k) > 1, angle(k) = angle(k) - 1

for angle(k) < -1, angle(k) = angle(k) + 1”

The frequency of the ramp output is controlled by a precision frequency generation algorithm that relies on the modulo nature of the finite length variables. The frequency of the output ramp signal is equal to

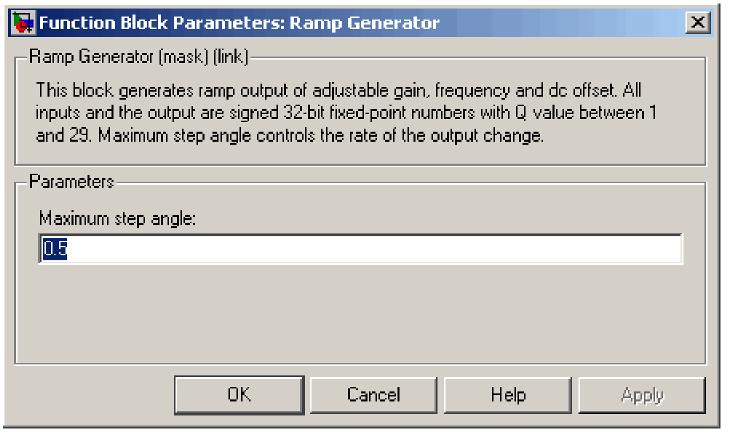
“f = (**Maximum step angle** * sampling rate) / 2^m ”

where m represents the fractional length of the data type of the inputs.

All math operations are carried out in fixed-point arithmetic, where the fixed-point fractional length is determined by the block's inputs.

Note To generate optimized code from this block, enable the `TI C28x` or `TI C28x (ISO)` TFL. See"About Target Function Libraries and Optimization for Embedded TargetsDesktop Targets".

Dialog Box



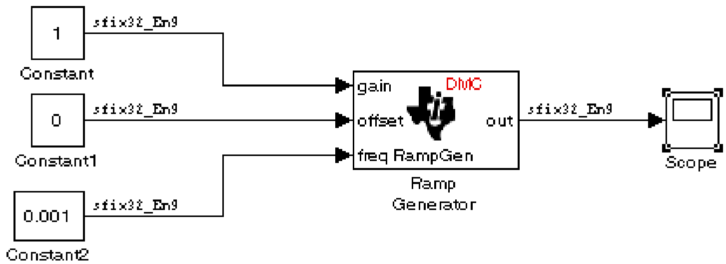
Maximum step angle

The maximum step size, which determines the rate of change of the output (i.e., the minimum period of the ramp signal).

When you enter double-precision floating-point values for parameters in the IQ Math blocks, the software converts them to single-precision values that are compatible with the behavior on c28x processor.

Examples

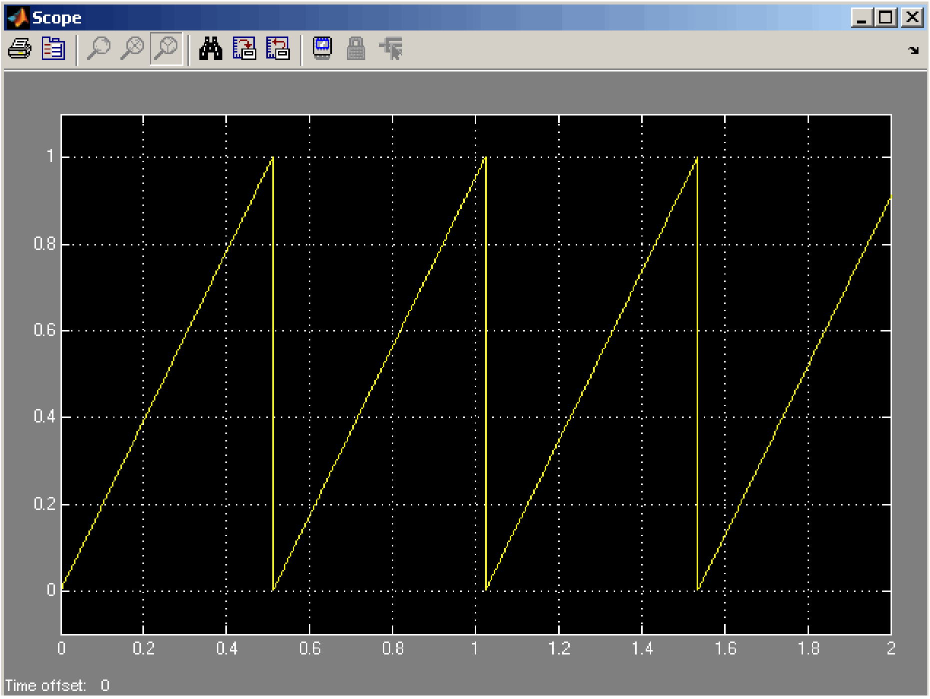
The following model demonstrates the Ramp Generator block. The Constant and Scope blocks are available in Simulink Commonly Used Blocks.



In your model, select **Simulation > Configuration Parameters**. On the **Solver** pane, set **Type** to `Fixed-step` and **Solver** to `Discrete (no continuous states)`. Set the parameter values for the blocks as shown in the following table.

Block	Connects to	Parameter	Value
Constant	Ramp Generator - `gain`	`Constant value`	`1`
		`Sample time`	`0.001`
		`Output data type`	`sfix(32)`
		`Output scalig value`	`2^-9`
Constant	Ramp Generator - `offset`	`Constant value`	`0`
		`Sample time`	`inf`
		`Output data type`	`sfix(32)`
		`Output scalig value`	`2^-9`
Constant	Ramp Generator - `freq`	`Constant value`	`0.001`
		`Sample time`	`inf`
		`Output data type`	`sfix(32)`
		`Output scalig value`	`2^-9`
C2000 Ramp Generator	Scope and Floating Scope (Simulink block)	`Maximum step angle`	`1`

When you run the model, the Scope block generates the following output (drag a zoom box around a portion of the output to change the display).



With fixed point calculations in IQMath, for a given frequency input on the block, **f_input**, the equation is:

“f = (Maximum step angle * f_input * sampling rate) / 2^m”

For example, if f_input = 0.001, the real value, 1, counts as fixed point with a fractional length of 9:

“f = (1 * 1 * (1/0.001)) / 2^9 = 1.9531 Hz”

Where 0.001 is the block sample time.

If we use normal math, and f_input is a non-fixed point real value, then:

“f = (Maximum step angle * f_input * sampling rate) / 1”

For example, if we are using floating point calculation:

“f = (1 * 0.001 * (1/0.001)) / 1 = 1 Hz”

When using fixed point with fractional length 9, the expected period becomes:

“$T = 1/f$ = 1/1.9531 Hz = 0.5120 s”

This result is what the above Scope output shows.

Note If you use different fractional lengths for the fixed point calculations, the output frequency varies depending on the precision.

See Also

C2000 Ramp Control

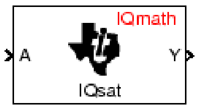
Purpose

Saturate IQ number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

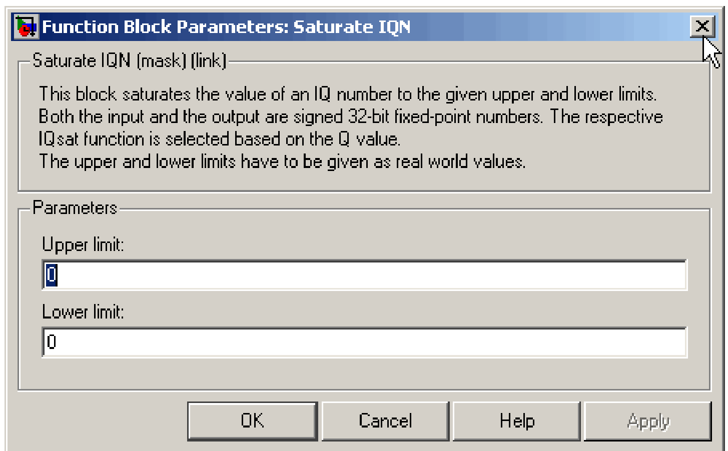


Saturate IQN

This block saturates an input IQ number to the specified upper and lower limits. The returned value is an IQ number of the same Q value as the input.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Upper Limit
Maximum real-world value to which to saturate

Lower Limit
Minimum real-world value to which to saturate

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Square Root IQN, C2000 Trig Fcn IQN

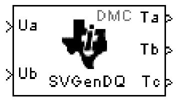
Purpose

Duty ratios for stator reference voltage

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description



Space Vector Generator

This block calculates appropriate duty ratios needed to generate a given stator reference voltage using space vector PWM technique. Space vector pulse width modulation is a switching sequence of the upper three power devices of a three-phase voltage source inverter and is used in applications such as AC induction and permanent magnet synchronous motor drives. The switching scheme results in three pseudosinusoidal currents in the stator phases. This technique approximates a given stator reference voltage vector by combining the switching pattern corresponding to the basic space vectors.

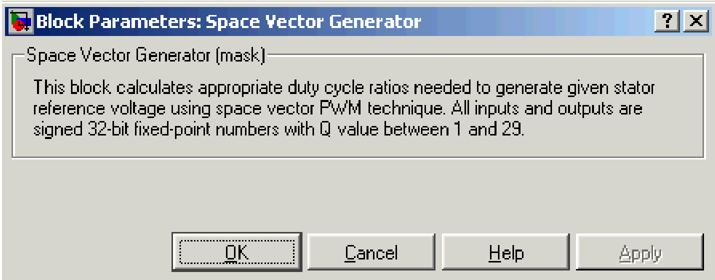
The inputs to this block are

- Alpha component — the reference stator voltage vector on the direct axis stationary reference frame (`Ua`)
- Beta component — the reference stator voltage vector on the direct axis quadrature reference frame (`Ub`)

The alpha and beta components are transformed via the inverse Clarke equation and projected into reference phase voltages. These voltages are represented in the outputs as the duty ratios of the PWM1 (`Ta`), PWM3 (`Tb`), and PWM5 (`Tc`).

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



References

For detailed information on the DMC library, see *C/F 28xx Digital Motor Control Library*, Literature Number SPRC080, available at the Texas Instruments Web site.

See Also

C2000 Clarke Transformation, C2000 Inverse Park Transformation, C2000 Park Transformation, C2000 PID Controller, C2000 Speed Measurement

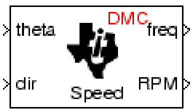
Purpose

Calculate motor speed

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x DMC

Description



Speed Measurement

This block calculates the motor speed based on the rotor position when the direction information is available. The inputs are the electrical angle (`theta`) and the direction of rotation (`dir`) from the encoder. The outputs are the speed normalized from 0 to 1 in the Q format (`freq`) and the speed in revolutions per minute (`rpm`).

Note

- To generate optimized code from this block, enable the `TI C28x` or `TI C28x (ISO)` TFL. See“About Target Function Libraries and Optimization for Embedded TargetsDesktop Targets”.
- This block does not call the corresponding Texas Instruments library function during code generation. Instead, the MathWorks code uses the TI functions global Q setting to adjust dynamically the Q format based on the block input. See “Using the IQmath Library” for more information.

Understanding the Theta Input to the Block

To indicate the rotational position of your motor, the block expects a 32-bit, fixed-point value that varies from 0 to 1.

Block input theta is defined by the following relations:

- A theta input signal equal to 0 indicates 0 degrees of rotation.
- A theta input signal equal to 1 indicates 360 degrees of rotation (one full rotation).

When the motor spins at a constant speed, theta (in counts) from your position sensor (encoder) should increase linearly from 0 to 1 and then abruptly return to 0, like a saw-shaped signal. Adjust the theta signal output from your encoder to get the correct input signal range for the Speed Measurement block. Then, convert your encoder signal to 32-bit fixed-point Q format that meets your resolution needs.

For example, if you are using a position sensor that generates 8000 counts for one full revolution of the motor, (0.0450 degrees per count), you need to reset your counter to 0 after your counter reaches 8000. Each time you read your encoder position, you need to convert the position to a 32-bit, fixed-point Q format value knowing that 8000 is represented as a 1.0. In this example your format could be Q31.

The Base Speed Parameter

Base speed is the maximum motor rotation rate to measure. This value is probably not the maximum speed the motor can achieve.

The Speed Measurement block calculates motor speed from two successive *theta* readings of the motor position, $theta_{new}$ and $theta_{old}$ (the base speed of the motor; and the time between readings). The maximum speed the block can calculate occurs when the difference between two successive samples [abs($theta_{new}$-$theta_{old}$)] is 1.0—one full motor revolution occurs between theta samples.

Therefore, the value you provide for the Base speed (in revolutions per minute) parameter is the speed, in revolutions per minute, at which your motor position signal reports one full revolution during one sample time. While the motor may spin faster than the base speed, the block cannot calculate the rotation rate correctly in that case. If the motor completes more than one revolution in one sample time, the calculated speed may be wrong. The block does not know that between samples $theta_{new}$ and $theta_{old}$, *theta* wrapped from 1 back to 0 and started counting up again.

The time difference between the two theta readings is the sample time. The Speed Measurement block inherits the sample time from the upstream block in your model. You set the sample time in the upstream block and then the Speed Measurement block uses that sample time to calculate the rotation rate of the motor.

The Sample Time Calculation

Motor speed measurements depend on the sample time you set in the model. Your sample time must be short enough to measure the full speed of the motor.

Two parameters drive your sample time—motor base speed and encoder counts per revolution. To be able to measure the maximum rotation rate, you must take at least one sample for each revolution. For a motor with base speed equal to 1000 rpm, which is 16.67 rps, you need to sample at 1/16.67 s, which is 0.06 s/sample. This sample rate of 16.67 samples per second is the maximum sample time (lowest sample rate) so that you can measure the full speed of the motor.

Using the same sample rate assumption, the minimum speed the block can measure depends on the encoder counts per revolution. At the minimum measurable motor speed, the encoder generates one count per sample period—16.67 counts per second. For an encoder that generates 8000 counts per revolution, this results in being able to measure a speed of [(16.67 counts/s) * (0.045 degrees/count)] = 0.752 degrees per second, or about 45 degrees per minute—one-eighth RPM.

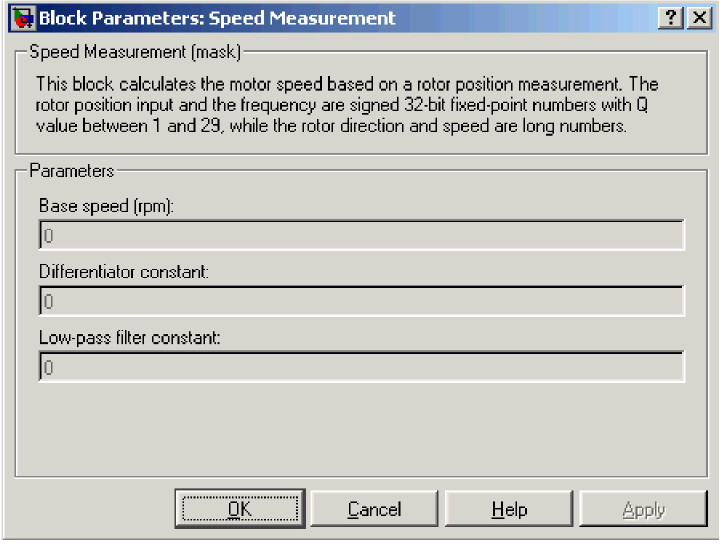
The Differentiator Constant

The differentiator constant is a scalar value applied to the block output. For example, setting it to `1` produces no effect on the output. Setting the constant to `1/4` multiplies the frequency and revolutions per minute outputs by 0.25. This setting can be useful when your motor has multiple pole pairs, and one electrical revolution is not equal to one mechanical revolution. The constant lets you account for the difference between electrical and mechanical rotation rates.

The Low-Pass Filter Constant

This block includes filtering capability if your position signal is noisy. Setting the filter constant to `0` disables the filter. Setting the filter constant to `1` filters out the entire signal and results in a block output equal to 0. Use a simulation to determine the best filter constant for your system. Your goal is to filter enough to remove the noise on your signal but not so much that the speed measurements cannot react to abrupt speed changes.

Dialog Box



Base speed

Maximum speed of the motor to measure in revolutions per minute.

Differentiator constant

Constant used in the differentiator equation that describes the rotor position.

Low-pass filter constant

Constant to apply to the lowpass filter. This constant is $1/(1+T*(2\pi f_c))$, where T is the sampling period and f_c is the cutoff frequency. The $1/(2\pi f_c)$ term is the lowpass filter time constant. This block uses a lowpass filter to reduce noise generated by the differentiator.

Example

The following example demonstrates how you configure the Speed Measurement block.

Configuring the Speed Measurement Block to Measure Motor Speed

Use the following process to set up the Speed Measurement block parameters.

1. Add the block to your model.
2. Open the block dialog box to view the block parameters.
3. Set the value for **Base Speed** to the maximum speed to measure, in revolutions per minute.
4. Enter values for **Differentiator** and **Low-Pass Filter Constant**.
5. Click **OK** to close the dialog box.

Setting the Sample Time to Measure Motor Speed

Use the following process to set the sample time for measuring the motor speed.

1. Open the block dialog box for the block before the Speed Measurement block in your model (the upstream or driving block).
2. Set the sample time parameter in the upstream block according to the sample time guidelines described in The Sample Time Calculation.
3. Click **OK** to close the dialog box.

References

For detailed information on the DMC library, see *C/F 28xx Digital Motor Control Library*, SPRC080, available at the Texas Instruments Web site.

See Also

C2000 Clarke Transformation, C2000 Inverse Park Transformation, C2000 Park Transformation, C2000 PID Controller, C2000 Space Vector Generator

C2000 Square Root IQN

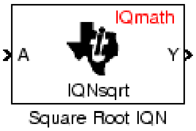
Purpose

Square root or inverse square root of IQ number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

Description

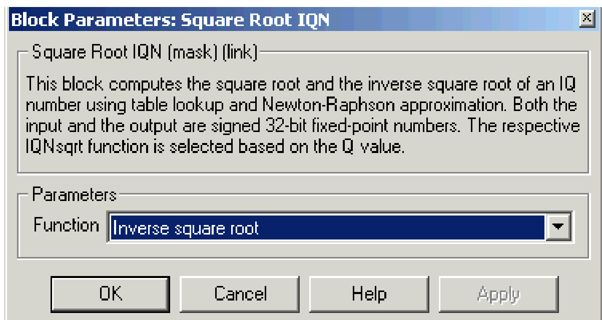


This block calculates the square root or inverse square root of an IQ number and returns an IQ number of the same Q format. The block uses table lookup and a Newton-Raphson approximation.

Negative inputs to this block return a value of zero.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Function

Whether to calculate the square root or inverse square root

- `Square root (_sqrt)` — Compute the square root.
- `Inverse square root (_isqrt)` — Compute the inverse square root.

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, Fractional part IQN x int32, C2000 Integer part IQN, Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Trig Fcn IQN

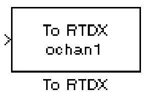
Purpose

Add RTDX communication channel to send data from target to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ RTDX Instrumentation

Description



Note This block will be removed from the Embedded Coder product in an upcoming release.

Note To use RTDX for C28x host/target communications, download and install TI DSP/BIOS. The DSP/BIOS installation includes files required for RTDX communications. For more information, see *DSP/BIOS, RTDX and Host-Target Communications*, Literature Number SPRA895, available at the Texas Instruments Web site.

When you generate code from Simulink in Simulink Coder software with a To RTDX block in your model, code generation inserts the C commands to create an RTDX output channel on the target DSP. The output channels transfer data from the target DSP to the host.

The generated code contains this command:

```
RTDX_enableOutput(&channelname)
```

where `channelname` is the name you enter in the **channelName** field in the To RTDX dialog box.

Note To RTDX blocks work only in code generation and when your model runs on your target. In simulations, this block does not perform any operations.

To use RTDX blocks in your model, you must do the following:

1 Add one or more To RTDX or From RTDX blocks to your model.

2 Download and run your model on your target.

3 Enable the RTDX channels from MATLAB or use **Enable RTDX channel on start-up** on the block dialog.

4 Use the `readmsg` and `writemsg` functions on the MATLAB command line to send and retrieve data from the target over RTDX.

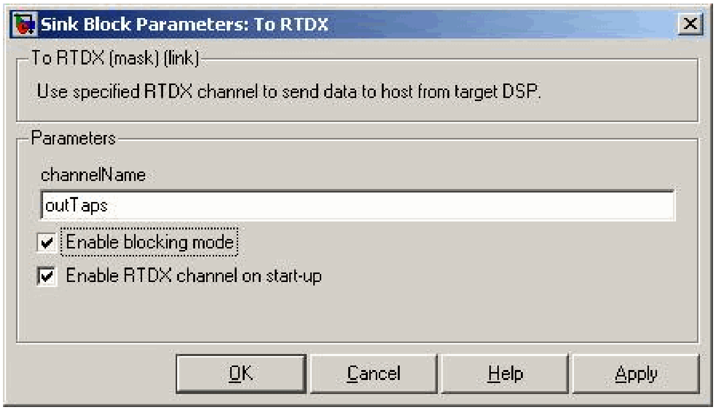
For more information about using RTDX in your model, see the following demos:

- Real-Time Data Exchange (RTDX™) Tutorial
- Comparing Simulation and Target Implementation with RTDX
- Real-Time Data Exchange via RTDX
- DC Motor Speed Control via RTDX™

Note To use RTDX with the XDS100 USB JTAG Emulator and the C28027 chip, add the following line to the linker command file:

```
_RTDX_interrupt_mask = ~0x000000008;
```

Dialog Box



Channel name
Name of the output channel to be created by the generated code. The channel name must meet C syntax requirements for length and character content.

Enable blocking mode
Enables blocking mode (selected by default). In blocking mode, writing a message is suspended while the RTDX channel is busy, that is, when data is being written in either direction. The code waits at the RTDX_write call site while the channel is busy. Any interrupt of the higher priority will temporary divert the program execution from this site, but it will eventually come back and wait until the channel stops writing.

When blocking mode is not enabled (when the check box is cleared), writing a message is abandoned if the RTDX channel is busy, and the code proceeds with the current iteration.

Enable RTDX channel on start-up
Enables the RTDX channel when you start the channel from MATLAB. With this selected, you do not need to use the `enable` function to prepare your RTDX channels. This option applies only to the channel you specify in **Channel name**. You do have to open the channel.

See Also

C2000 From RTDX

References

RTDX 2.0 User's Guide, Literature Number: SPRUFC7, available from the Texas Instruments Web site.

How to Write an RTDX Host Application Using MATLAB, Literature Number: SPRA386, available from the Texas Instruments Web site.

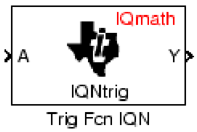
Purpose

Sine, cosine, or arc tangent of IQ number

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Optimization/ C28x IQmath

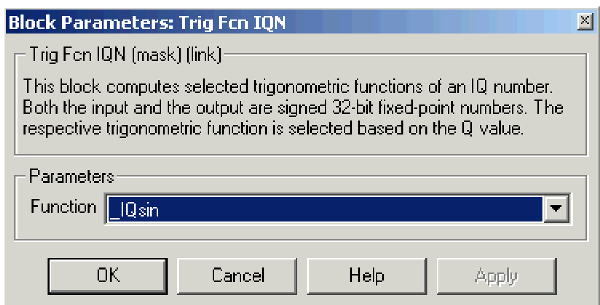
Description



This block calculates basic trigonometric functions and returns the result as an IQ number. Valid Q values for `_IQsinPU` and `_IQcosPU` are 1 to 30. For all others, valid Q values are from 1 to 29.

Note The implementation of this block does not call the corresponding Texas Instruments library function during code generation. The TI function uses a global Q setting and the MathWorks code used by this block dynamically adjusts the Q format based on the block input. See "Using the IQmath Library" for more information.

Dialog Box



Function

Type of trigonometric function to calculate:

- `_IQsin` — Compute the sine `(sin(A))`, where A is in radians.
- `_IQsinPU` — Compute the sine per unit `(sin(2*pi*A))`, where A is in per-unit radians.
- `_IQcos` — Compute the cosine `(cos(A))`, where A is in radians.

- `_IQcosPU` — Compute the cosine per unit `(cos(2*pi*A))`, where A is in per-unit radians.

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

See Also

C2000 Absolute IQN, C2000 Arctangent IQN, C2000 Division IQN, C2000 Float to IQN, C2000 Fractional part IQN, C2000 Fractional part IQN x int32, C2000 Integer part IQN, C2000 Integer part IQN x int32, C2000 IQN to Float, C2000 IQN x int32, C2000 IQN x IQN, C2000 IQN1 to IQN2, C2000 IQN1 x IQN2, C2000 Magnitude IQN, C2000 Saturate IQN, C2000 Square Root IQN

C5510 DSK ADC

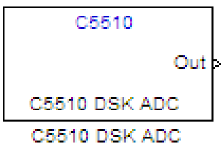
Purpose

Configure AIC23 and peripherals to collect data from analog jacks and output digital data

Library

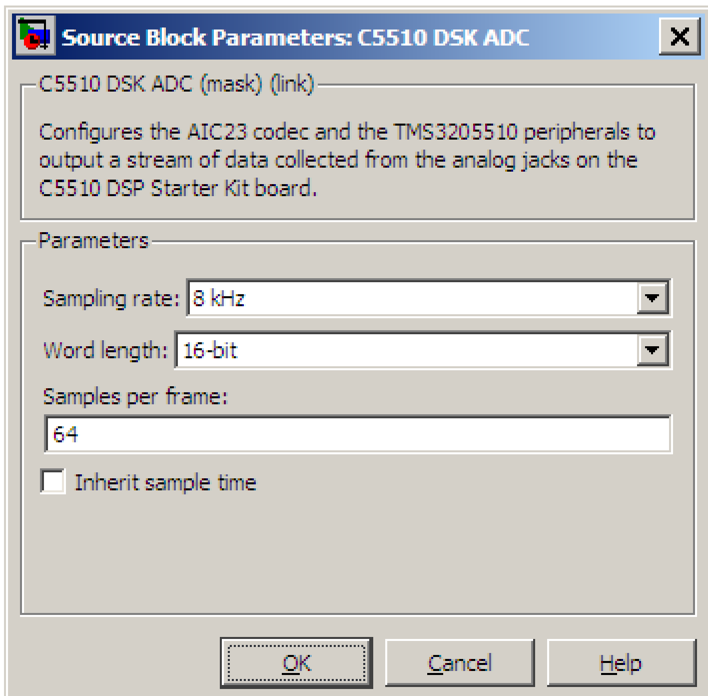
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ C5510 DSK

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

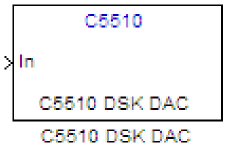
Purpose

Configure AIC23 codec and peripherals to send data stream to output jack

Library

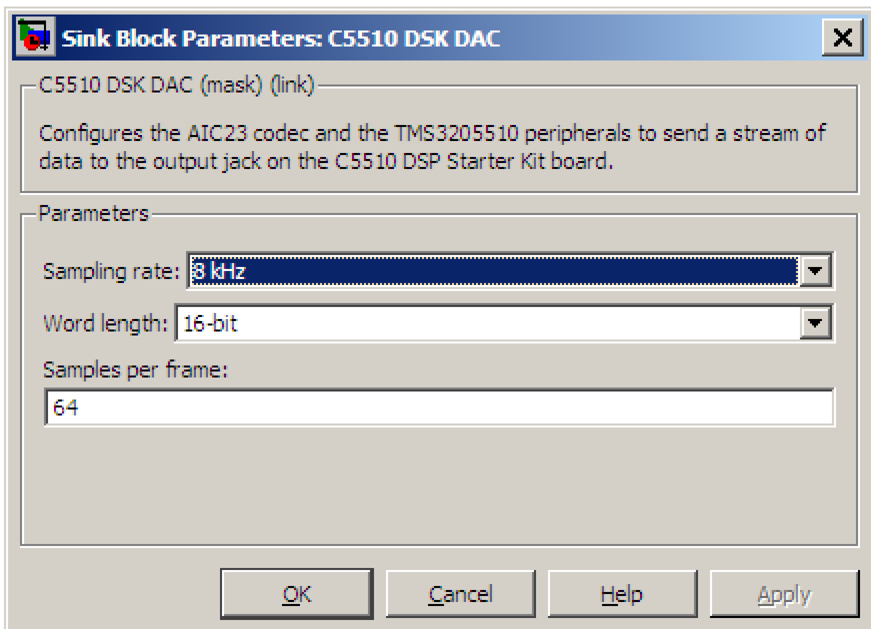
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ C5510 DSK

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 do not use an accurate setting, the DAC cannot convert the data correctly.

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

Purpose

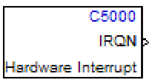
Interrupt Service Routine to handle hardware interrupt on C5000 and C6000 processors

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

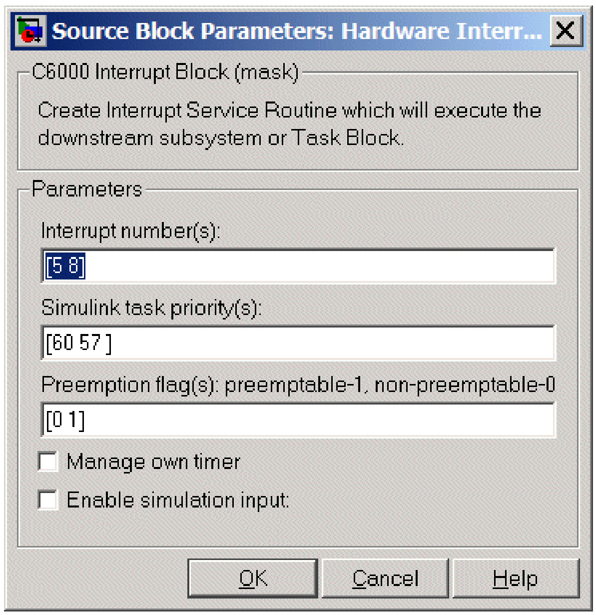
Description



Hardware Interrupt

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 the proper 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.

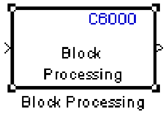
Purpose

Repeat user-specified operation on submatrices of input matrix, using internal memory of DSP for increased efficiency

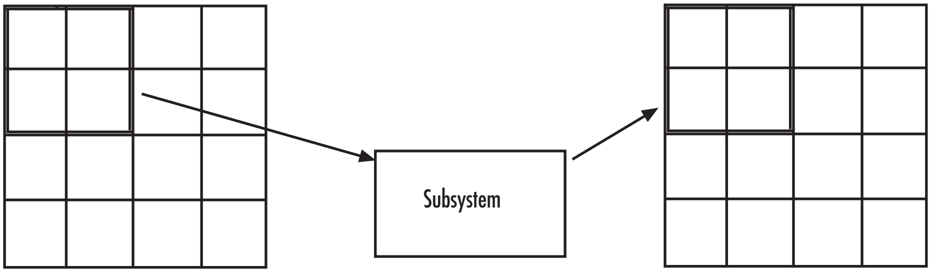
Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



Using Direct Memory Access (DMA) on the processor, the Block Processing block extracts submatrices of a user-specified size from each input matrix. It sends each submatrix to a subsystem for processing, and then reassembles each subsystem output into the output matrix, as shown in the following figure. While processing images as matrices, this submatrix capability can greatly improve the throughput.



Note Because you modify the Block Processing block subsystem, the link between this block and the block library is broken when you click-and-drag a Block Processing block into your model. Thus, this block is not automatically updated if you upgrade to a newer version of the Embedded Coder. To delete blocks from this subsystem without triggering a warning, right-click on the block and select **Look under mask**. If you search for library blocks in a model, this block is not part of the results.

The blocks inside the subsystem dictate the following block configuration information:

- Frame status of the input and output signals
- Whether the block supports single channel or multichannel signals
- Which data types this block supports

Use the **Number of inputs** and **Number of outputs** parameters to specify the number of input and output ports on the Block Processing block.

Use the **Block size** parameter to specify the size of each submatrix in cell array format. Each vector in the cell array corresponds to one input; the block uses the vectors in the order you enter them. If you have one input port, enter one vector. If you have more than one input port, you can enter one vector that is used for all inputs or you can specify a different vector for each input. For example, to specify each submatrix as a 2-by-3 array, enter `{[2 3]}`. The output matrix size depends on the size of the submatrix at the output of the subsystem and the number of submatrices at the input. For example, if the output submatrix size is 32x16 and the input submatrix sizes are 8x16, the total output matrix size will be 256x256. If the block size specified does not subdivide an input matrix evenly, i.e. there are leftover matrix elements which are not covered by the subdivision, those uncovered elements will be ignored.

Use the **Overlap** parameter to specify the overlap of each submatrix in cell array format. Each vector in the cell array corresponds to the overlap of one input; the block uses the vectors in the order they are specified. If you enter one vector, each overlap is the same size. For example, to specify that each 3-by-3 submatrix overlap by 1 row and 2 columns, enter `{[1 2]}`.

The **Traverse order** parameter determines how the block extracts submatrices from the input matrix. If you select `Row-wise`, the block extracts submatrices by moving across the rows. If you select `Column-wise`, the block extracts submatrices by moving down the columns.

Click **Open Subsystem** to open the block subsystem. Click-and-drag blocks into this subsystem to define the processing operations the block performs on the submatrices. The input to this subsystem are the submatrices defined by the **Block size** parameter.

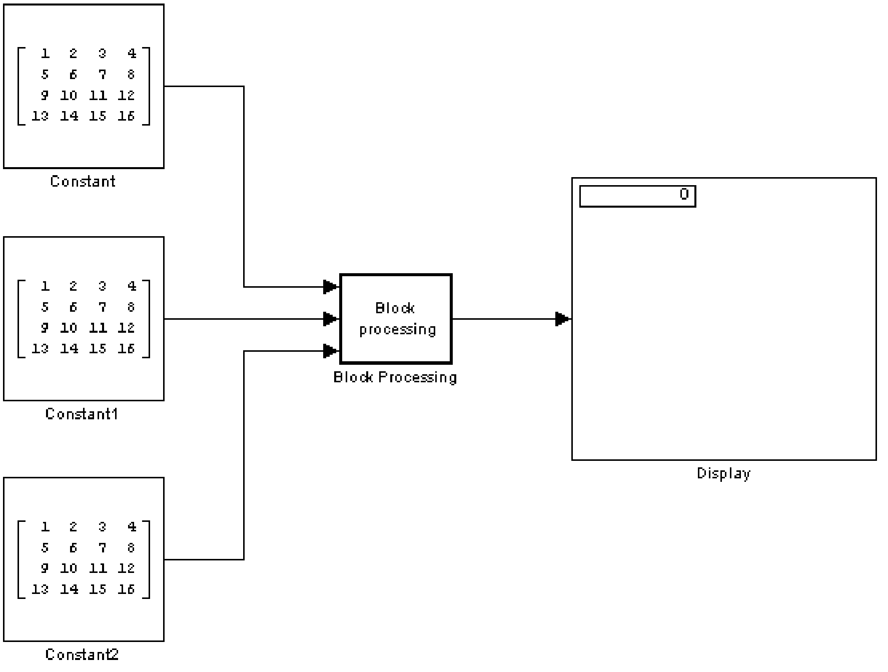
Note When you place an Assignment block inside a Block Processing block subsystem, the Assignment block behaves as though it is inside a For Iterator block. For a description of this behavior, refer to "Iterated Assignment" on the Assignment block reference page. To produce the normal behavior of the Assignment block, use an Overwrite Values block inside the Block Processing block subsystem.

Example

This section provides an example that applies the block processing block to multiply and add submatrices.

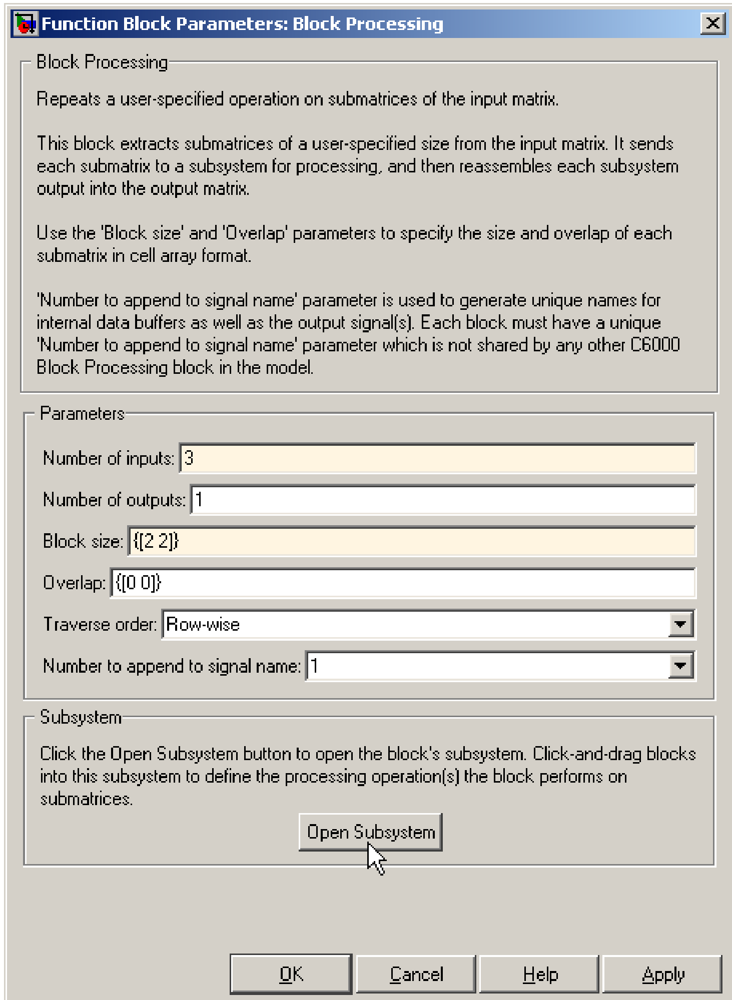
Multiple Inputs

In this example, you multiply each element of three input matrices by two and add the results using the Block Processing block. Suppose you have the following model:



1 Use the Block Processing block to perform the multiplication and addition on submatrices of the three input matrices. Set the block parameters as shown in the following figure:

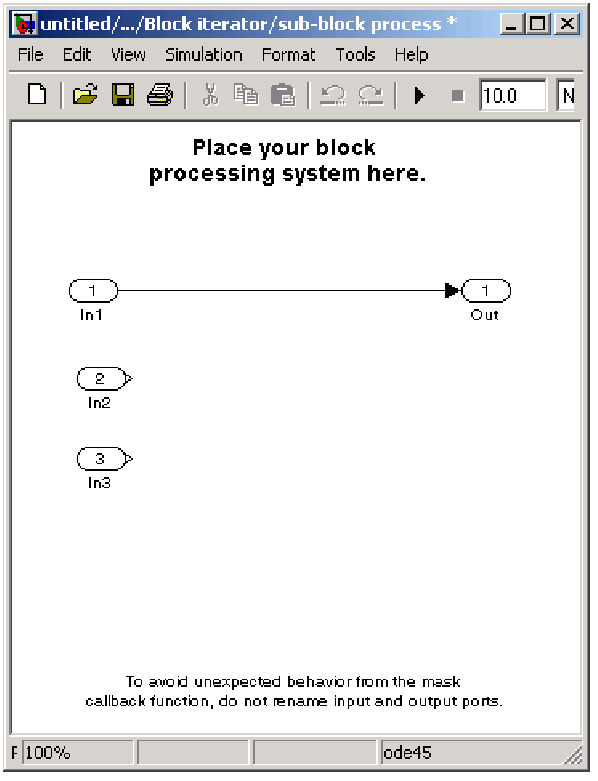
- **Number of inputs** = 3
- **Number of outputs** = 1
- **Block size** = {[2 2]}



For each iteration, the block sends a 2-by-2 submatrix from each input matrix to the Block Processing block subsystem to be processed. The block calculates its total number of iterations using the dimensions of the matrix connected to the top input port. In this case, the first input is a 4-by-4 matrix. The block can extract four 2-by-2 submatrices from this input matrix, so the block iterates four times.

2 Click **Open Subsystem**.

The block subsystem opens.



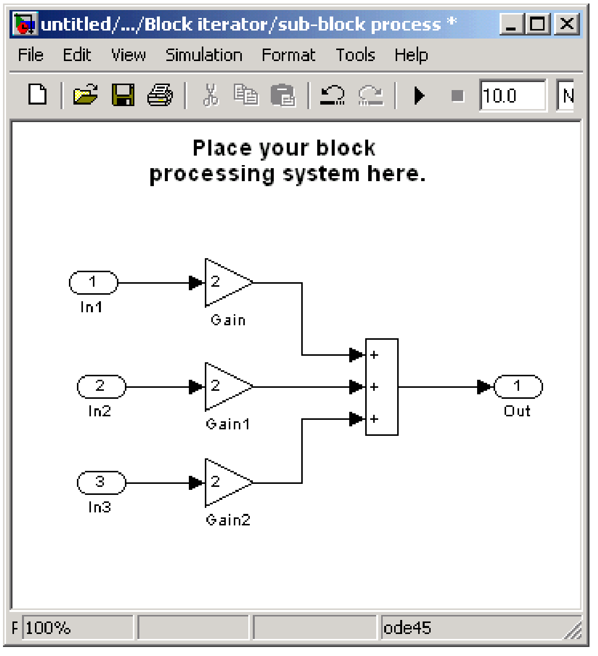
3 Click and drag the blocks shown in the following table into the subsystem.

Block	Library	Quantity
Gain	Simulink / Math Operations	3
Sum	Simulink / Math Operations	1

4 Use the Gain blocks to multiply the elements of each submatrix by two. Set the **Gain** parameter to `2`.

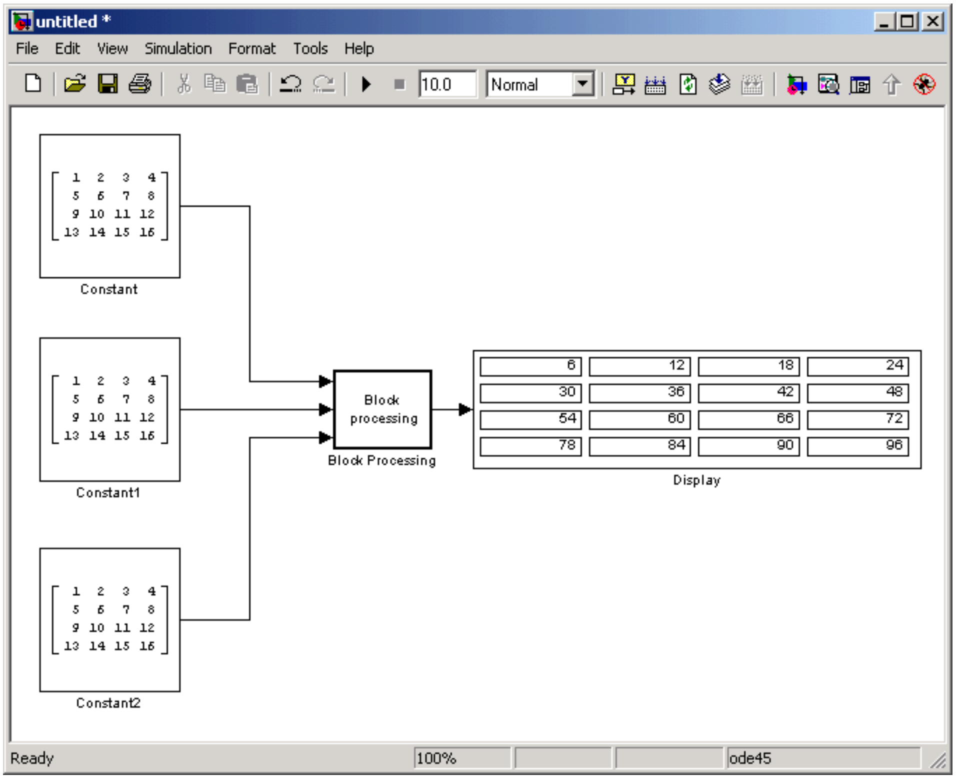
5 Use the Sum block to add the values. Set the **Icon shape** parameter to `rectangular` and the **List of signs** parameter to +++.

6 Connect the blocks as shown in the following figure.



7 Close the subsystem and click **OK**.

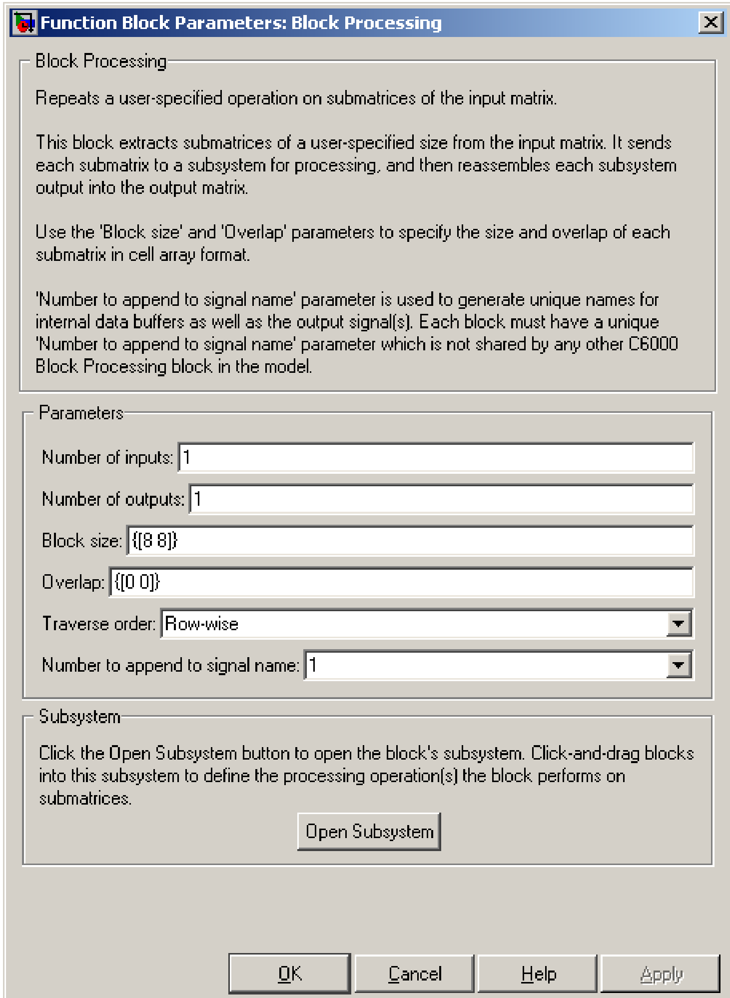
8 Run the model.



The Block Processing block operates on the submatrices, assembles the results into an output matrix, and then uses the Display block to present the output matrix.

Dialog Box

The Block Processing dialog box appears as shown in the following figure.



Number of inputs
Enter the number of input ports on the Block Processing block.

Number of outputs
Enter the number of output ports on the Block Processing block.

Block size
Specify the size of each submatrix in cell array format. Each vector in the cell array corresponds to one input.

Overlap
Specify the overlap of each submatrix in cell array format. Each vector in the cell array corresponds to the overlap of one input.

Traverse order
Determines how the block extracts submatrices from the input matrix. If you select `Row-wise`, the block extracts submatrices by moving across the rows. If you select `Column-wise`, the block extracts submatrices by moving down the columns.

Open Subsystem
Click this button to open the block's subsystem. Click and drag blocks into this subsystem to define the processing the block performs on the submatrices.

See Also

Memory Allocate, Memory Copy, C6000 EDMA

Purpose

Separate interleaved YCbCr 4:2:2 data into Y, Cb, and Cr components

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Avnet S3ADSP DM6437

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description

Deinterleave

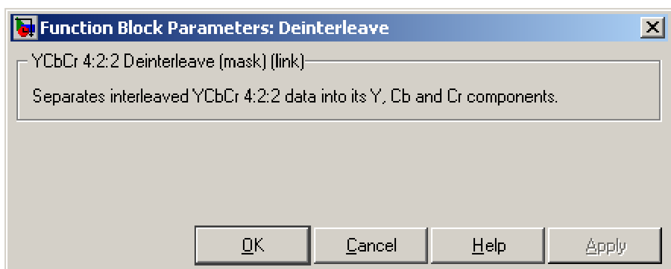
This block separates interleaved YCbCr 4:2:2 data into its luma component (Y'), blue-difference chroma component (Cb), and red-difference chroma component (Cr).

The input, YCbCr, is a (2*M)*N array of 8-bit unsigned values representing an interleaved YCbCr 4:2:2 image where the size of the luma plane, Y, is M*N. Input data is assumed to be in row-major format, and the data stored in each row of the input is assumed to be interleaved in the following order:

```
Cb(1), Y(1), Cr(1), Y(2), ......  Cb(M), Y(M), Cr(M), Y(M)
```

The deinterleaved outputs are the planar format luma component, Y, and the chroma components, Cb and Cr, of the YCbCr 4:2:2 input. If the input image is a (2*M) by N matrix, then the output dimensions for the Y port is (M*N) and the dimensions for the Cb and Cr ports are (M/2) by N.

Dialog Box



This block does not have settable options.

See Also

C6000 Interleave

C6000 EDMA

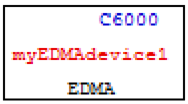
Purpose

Configure EDMA Controller on C6000 processor

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



EDMA

Use this block to configure the Enhanced Direct Memory Access (EDMA) Controller on C6000 processors. The controller manages data transfers between the device peripherals on the C6000 processors and the level two (L2) cache/memory controller. Data transfers handled by the controller include:

- Host accesses to cache
- Accessing noncacheable memory
- Servicing cache
- Transferring data by user programs

EDMA controller handles transfers without involving the processor and can process transfers between any addressable memory spaces, including internal and external memory.

For details about the EDMA controller, refer to *TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide*, SPRU234, from the Texas Instruments Web site.

Note The C6000 EDMA block does not support C64x+ processors, such as the C6455 or TCI6482.

EDMA blocks provide two operating modes—open an EDMA channel and allocate a table in EDMA parameter RAM (PaRAM).

The open channel mode opens an EDMA channel for the controller. When you open a channel, EDMA sets the transfer parameters for the channel and writes those to a table as PaRAM entries.

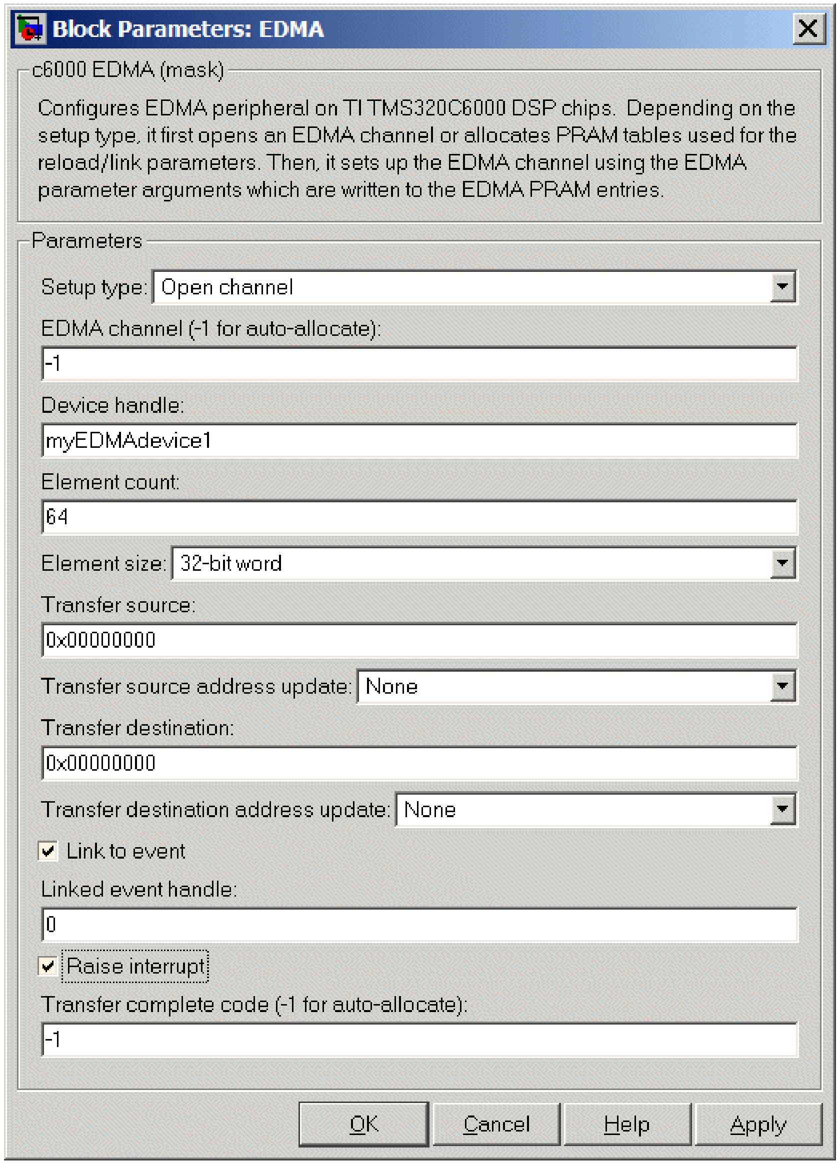
In allocate table mode, the block sets the EDMA transfer parameters and places them in a table in EDMA PaRAM without opening a channel. With this mode, you can use EDMA channels and transfers to develop complex memory structures like sorting, or circular buffers. The allocate table operating mode lets you link multiple EDMA blocks on one EDMA channel. One EDMA block opens an EDMA channel and succeeding blocks link to the open channel and originating EDMA block by the device handle setting.

Use the following procedure to link EDMA blocks in a model:

1. Add an EDMA block to your model, open the block dialog box, and set **Setup type** to `Open channel`.
2. Assign an EDMA channel to use in **EDMA channel (-1 for auto-allocate)** by entering a channel number or entering `-1` to let the block choose the channel.
3. In **Device handle**, provide a name for this EDMA block. The name you enter becomes the block identifier for other blocks to link to this block. Use any valid C variable string.
4. Close the block dialog box.
5. Add a second EDMA block to your model, and open the block dialog box to set the block parameters.
6. Select `Allocate table` from the **Setup type** list.
7. Select the **Link to event** check box.
8. Enter the device handle from the earlier block to link to in **Linked event handle** in this block. The two blocks are linked together through the device handle and they use the same channel.
9. Close the block dialog box.
10. To link more EDMA blocks to this channel, repeat steps 5 through 9 for each new block, entering the same device handle.

For a demonstration of using and linking EDMA blocks, refer to the demo Custom Device Driver via Legacy Code Integration in the Embedded Coder demos in the online help system.

Dialog Box


Block Parameters: EDMA
c6000 EDMA (mask)
Configures EDMA peripheral on TI TMS320C6000 DSP chips. Depending on the setup type, it first opens an EDMA channel or allocates PRAM tables used for the reload/link parameters. Then, it sets up the EDMA channel using the EDMA parameter arguments which are written to the EDMA PRAM entries.
Parameters
Setup type:
Open channel
EDMA channel (-1 for auto-allocate):
-1
Device handle:
myEDMAdevice1
Element count:
64
Element size:
32-bit word
Transfer source:
0x00000000
Transfer source address update:
None
Transfer destination:
0x00000000
Transfer destination address update:
None
Link to event
Linked event handle:
0
Raise interrupt
Transfer complete code (-1 for auto-allocate):
-1
OK
Cancel
Help
Apply

The preceding dialog box shown presents all of the parameters available. In some cases, parameters are available only when you select other parameters. The following list of block parameters describes all of the available parameters for the block and when one parameter enables another.

Setup type

Choose either `Open channel` or `Allocate table` from the list. If this is the only EDMA block in your model, choose `Open channel`. If your model includes multiple EDMA blocks, choose `Open channel` when each block should use a different channel. Select `Allocate table` for any block that you plan to link to another EDMA block.

EDMA channel (-1 for auto-allocate)

Enter an integer from 0 to 63 to specify the EDMA channel to use. If you enter `-1`, the block assigns the channel automatically from the available channels.

Device handle

Provide a name for this block. The name you enter must be a valid C variable. The EDMA controller uses the name as the identifier for this block and open channel. Other EDMA blocks in your model can link to this block and channel by using the device handle you enter.

Element count

Specifies the number of elements in a frame. The value 65355 is the maximum number of elements allowed in one frame. The value defaults to `64` elements.

Element size

EDMA supports 32-bit words, 16-bit half words, and 8-bit bytes. Select one of the list entries according to your needs.

Transfer source

Enter the address of the elements to transfer. Specify the address as a hexadecimal value as shown by the default address `0x.00000000`

Transfer source address update

Select whether to enable transfer source update on the EDMA controller. When you select an option from the list, the controller updates the transfer source address according to your choice. Choose one of the list entries shown in the following table.

Option	Effect on Transfer Source Address	Condition Indicated
None	Does not change address after submitting the transfer request.	Indicates that all of the elements to transfer are located at the same address in memory.
Increment	Increases the transfer address by the value in **Element count** after submitting the transfer request.	Indicates that the elements are contiguous, with each subsequent element located at a higher address than the previous element.
Decrement	Decreases the transfer address by the value in **Element count** after submitting the transfer request.	Indicates that the elements are contiguous, with each subsequent element located at a lower address than the previous element.

Transfer destination

Enter the destination memory address for the data transfer. Specify the address as a hexadecimal value as shown by the default address `0x.00000000`

Transfer destination address update

Select whether to enable transfer destination update on the EDMA controller. When you select an option from the list, the controller updates the transfer destination address according to your choice. Choose one of the list entries shown in the following table.

Option	Effect on Transfer Destination Address	Condition Indicated
`None`	Does not change address after submitting the transfer request.	Indicates that all of the elements to transfer are located at the same address in memory.
`Increment`	Increases the transfer address by the value in **Element count** after submitting the transfer request.	Indicates that the elements are contiguous, with each subsequent element located at a higher address than the previous element.
`Decrement`	Decreases the transfer address by the value in **Element count** after	Indicates that the elements are

Option	Effect on Transfer Destination Address	Condition Indicated
	submitting the transfer request.	contiguous, with each subsequent element located at a lower address than the previous element.

Link to event

You can link EDMA transfers together to create more complicated memory applications such as buffers and sorting routines. When you select **Link to event** to enable linking, the EDMA controller link feature reloads the current transfer parameters from PaRAM when the previous transfer is complete.

Linked event handle

To link to another EDMA block to create more complex memory applications, enter the device handle from the EDMA block to link to in **Linked event handle**. This entry is an alphanumeric string and the EDMA controller interprets your entry as a string.

Raise interrupt

Select this check box to direct the EDMA controller to raise an interrupt when the transfer request completes. When you select this parameter, you enable the Transfer complete code (-1 for auto-allocate) option. Clearing Raise interrupt stops the controller from raising the interrupt on TR completion.

Transfer complete code (-1 for auto-allocate)

The transfer code Indicates when the controller has submitted a required number of transfer requests (TR). Provide an integer from 0 and 62. On C67x processors, the code must be from 0 to 15. The default value of -1 lets the controller assign the transfer code for this channel.

When you enable this option, the EDMA controller submits the transfer request with a request that the controller signal completion of the transfer with this code. When the transfer is completed, the transfer controller returns the specified code to the EDMA controller.

After the EDMA controller receives the transfer complete code in response to the TR, the controller uses the code to trigger another TR or to raise an interrupt to the processor when you select **Raise interrupt**.

References

For details about the EDMA controller, refer to *TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide*, SPRU234, available from the Texas Instruments Web site.

For an introduction to the EDMA controller, refer to *TMS320C6000 Peripherals Reference Guide*, SPRU190, which provides an overview of the controller, available from the Texas Instruments Web site.

See Also

Memory Allocate, Memory Copy

Purpose

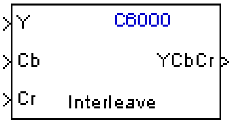
Convert planar YCbCr 4:2:2 data to interleaved YCbCr 4:2:2 data

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Avnet S3ADSP DM6437

Description

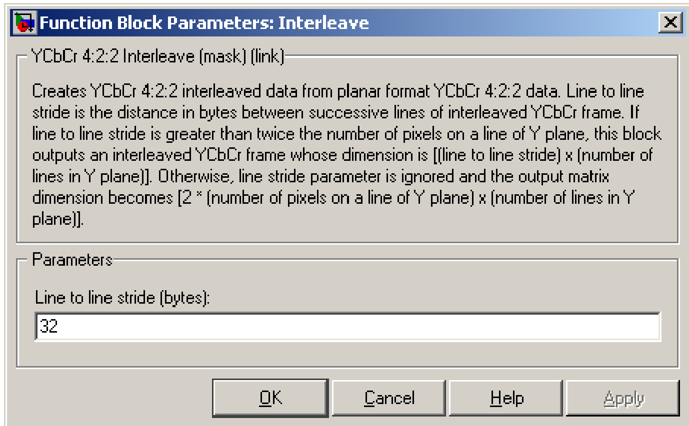


This block takes planar YCbCr 4:2:2 data on three separate inputs and converts them to a single interleaved YCbCr 4:2:2 data output.

The input is a planar, color separated, YCbCr 4:2:2 image represented as a 2-D matrix of 8-bit unsigned integers. There are three input ports, one each for the luma component (Y), blue-difference chroma component (Cb), and red-difference chroma component (Cr). If the input to the Y port has dimensions M*N, the input to the Cb and Cr ports must be (M/2) by N.

The output is an interleaved YCbCr 4:2:2 image represented as a 2-D matrix of 8-bit unsigned integers. If the dimension of the Y port is M*N and dimensions of the Cb and Cr ports are M/2 by N, the image dimensions of the YCbCr output dimensions are 2*M*N under normal conditions. If you specify a line-to-line stride greater than 2*M in the block's mask, the output dimensions become (line-to-line stride)*N.

Dialog Box



Line to line stride (bytes)

Use the line-to-line stride parameter to satisfy the input requirements of the DM6437EVM Video Display block. Because of hardware requirements, each line of the input to the DM6437EVM Video Display block must have a size that is multiple of 32 bytes. For example, if the image you want to display is 180 by 120, use a line-to-line stride of 384 to satisfy the hardware requirements. Under normal conditions, the output of the Interleave block would have size 360x120 which would not be accepted by the DM6437EVM Video Display block. By using a line stride of 384, the block outputs a 384 by 120 matrix—of which only the 360x120 portion contains valid data—that is readily accepted by the DM6437EVM Video Display block.

Line-to-line stride is the distance in bytes between successive lines of an interleaved YCbCr frame. If line-to-line stride is greater than twice the number of pixels on a line of Y plane, this block outputs an interleaved YCbCr frame whose dimensions are the line-to-line stride times the number of lines in Y plane. Otherwise, line stride parameter is ignored, and the output matrix dimension becomes 2*(number of pixels on a line of Y plane)*(the number of lines in Y plane).

See Also

C6000 Deinterleave

C6000 IP Config

Purpose

Configure Internet Protocol on C6000 targets with Ethernet ports

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Avnet S3ADSP DM6437

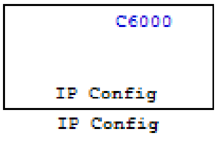
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6747 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM648 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ 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.

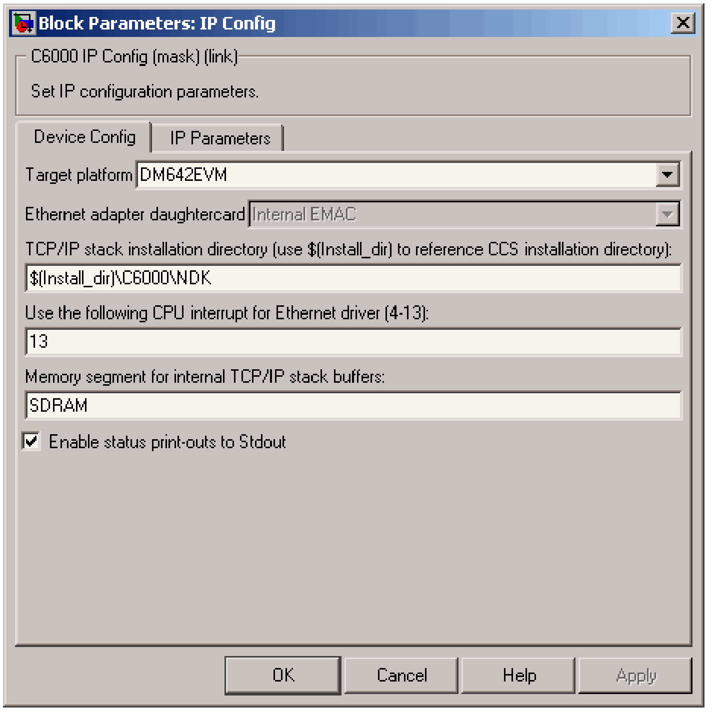
Whether you choose to use dynamic addressing, you must set the Host name, and select and set the **Use the following CPU interrupt for Ethernet driver (4-13)** options.

When you build and run your model, this block has no effect. 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 appropriate target board from the list. Changing the target platform changes the entry on the **Ethernet adapter daughtercard** list.

Ethernet adapter daughtercard

After you select you target platform, this option lets you select whatever daughtercard is available to implement Ethernet communications on the target.

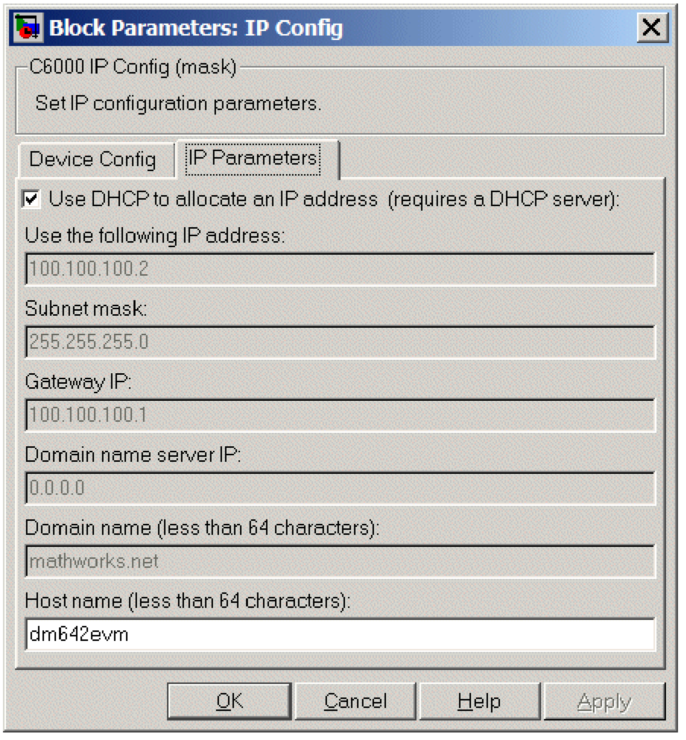
TCP/IP stack installation folder
To use the UDP and TCP blocks for the board, you must install the TMS320C6000 TCP/IP Stack from Texas Instruments. Specify the folder where the TMS320C6000 TCP/IP Stack from Texas Instruments is installed.

Use the following CPU interrupt for Ethernet driver (4-13)
The Ethernet driver on the DM642 can respond to any one of the CPU interrupts 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 Preferences block in your model.

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



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 correctly. 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 correct 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

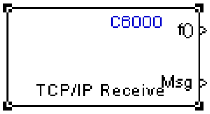
Purpose

Receive message from remote IP interface

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ 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. There is no restriction on message size.

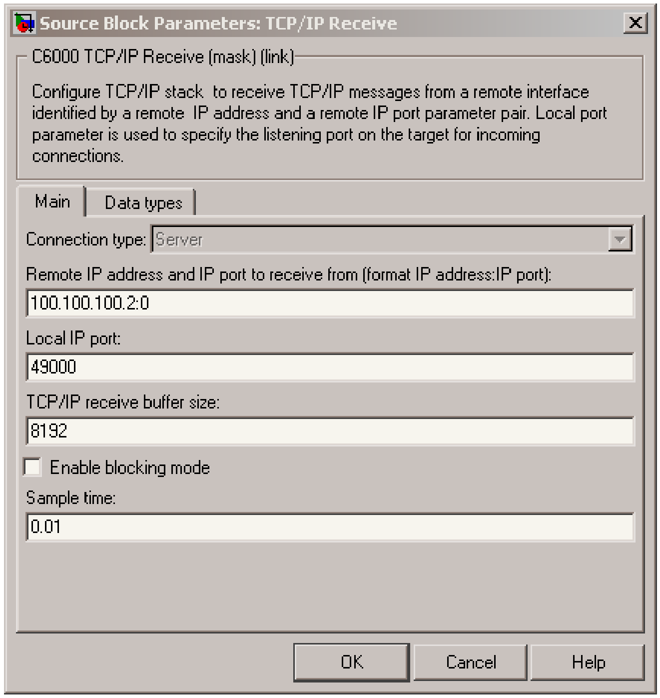
A second block output is a function call port that issues a function call whenever a new message is available on the receive buffer.

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. Any external TCP/IP interface that sends 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.

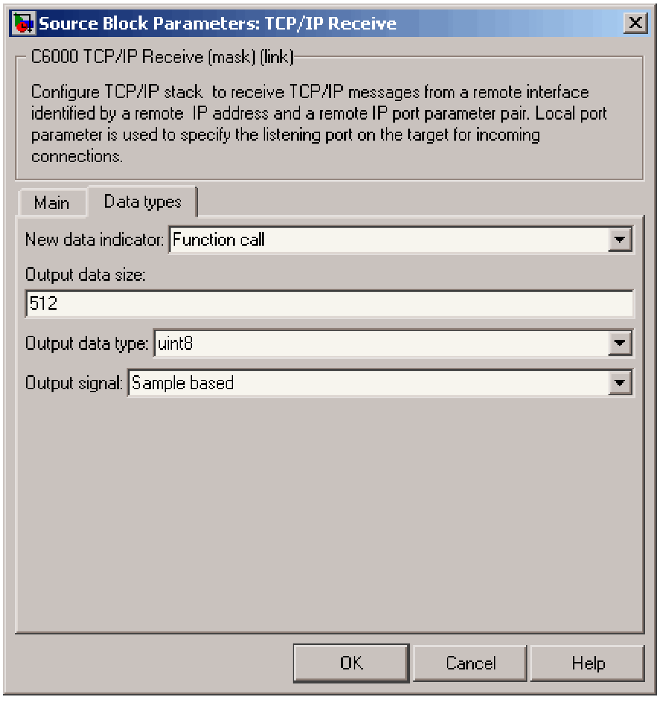
The block receives or outputs data at any time. Processes do not wait for data. Disabling blocking activates the **Sample time** parameter and adds an additional function call port to the block that indicates when the data port contains new, valid data.

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 any 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

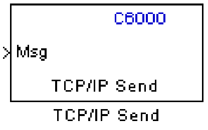
Purpose

Send message to remote IP interface

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Target Communication

Description



Adding this block to your Simulink model results in generated code that configures TCP/IP on your target to send messages.

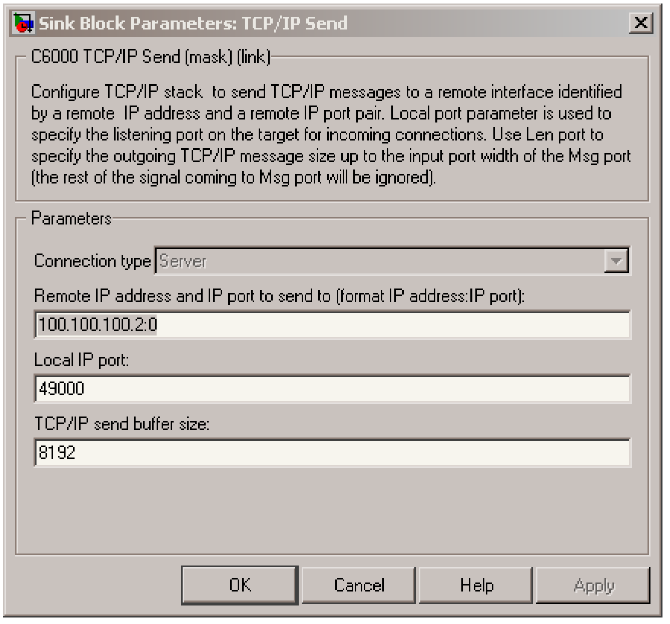
To use this block with the C6416, C6713, or C6713 DSK targets, you must meet the following requirements.

- 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. There is no restriction on the data type of the message to be sent, as long as it is a built-in Simulink data type. There is also no restriction on the size of the data to be transmitted.

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

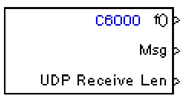
Purpose

Receive `uint8` vector as UDP message

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Target Communication

Description



UDP Receive

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.

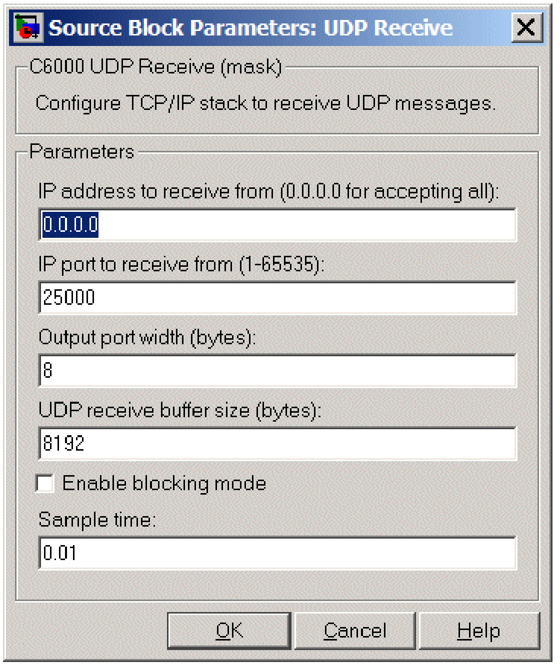
In non blocking mode, the data in the Msg port is not valid unless the block issues a function call.

C6000 UDP Receive blocks operate only to generate code for the target Ethernet driver. They do not perform any 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 any IP address. 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 be able to set IP address to receive from to any value except for 0.0.0.0. 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 as large or larger than any message 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 receives data or sends data at any time. 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 always 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

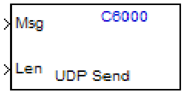
Purpose

Send UDP message to host

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Target Communication

Description



UDP Send

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.

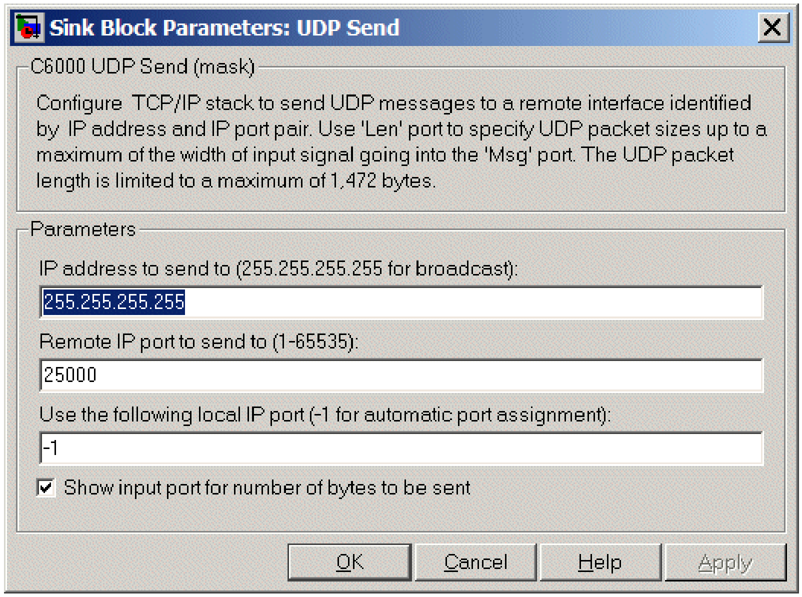
- Install the D.signT DSK-91C111 Ethernet adapter daughter card.
- Install the Texas Instruments TMS320C6000 TCP/IP stack software.

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 any 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 any function in 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 any 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 if necessary.

See Also

C6000 TCP/IP Receive, C6000 TCP/IP Send, C6000 UDP Receive

C62x Autocorrelation

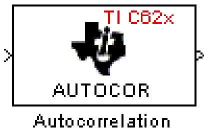
Purpose

Autocorrelate input vector or frame-based matrix

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

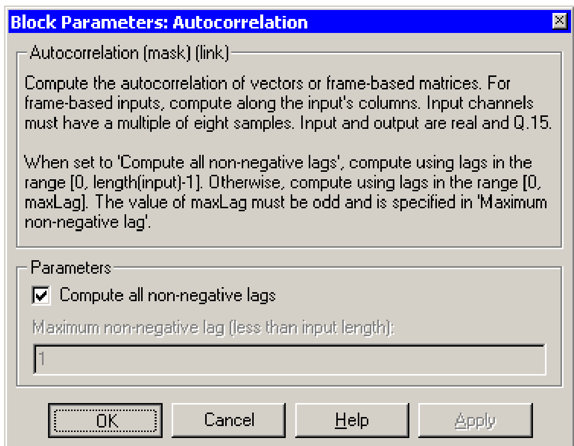
Description



The Autocorrelation block computes the autocorrelation of an input vector or frame-based matrix. For frame-based inputs, the autocorrelation is computed along each of the input's columns. The number of samples in the input channels must be an integer multiple of eight. Input and output signals are real and Q.15.

Autocorrelation blocks support discrete sample times and little-endian code generation only.

Dialog Box



Compute all non-negative lags

When you select this parameter, the autocorrelation is performed using all nonnegative lags, where the number of lags is one less than the length of the input. The lags produced are therefore in the range [0, `length`(input)-1]. When this parameter is not selected, you specify the lags used in **Maximum non-negative lag (less than input length)**.

Maximum non-negative lag (less than input length)
Specify the maximum lag (maxLag) the block should use in performing the autocorrelation. The lags used are in the range [0, maxLag]. The maximum lag must be odd. Enable this parameter by clearing the **Compute all non-negative lags** parameter.

Algorithm

In simulation, the Autocorrelation block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_autocor`. During code generation, this block calls the `DSP_autocor` routine to produce optimized code.

C62x Bit Reverse

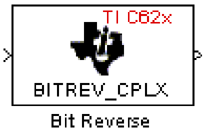
Purpose

Bit-reverse elements of each complex input signal channel

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

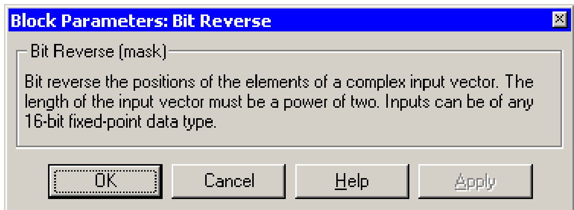
Description



The Bit Reverse block bit-reverses the elements of each channel of a complex input signal, X. The Bit Reverse block is primarily used to provide correctly-ordered inputs and outputs to or from blocks that perform FFTs. Inputs to this block must be 16-bit fixed-point data types.

The Bit Reverse block supports discrete sample times and little-endian code generation only.

Dialog Box

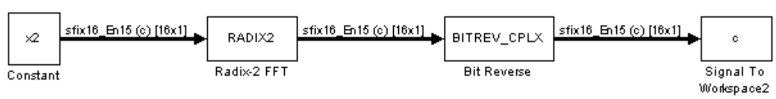


Algorithm

In simulation, the Bit Reverse block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_bitrev_cplx`. During code generation, this block calls the `DSP_bitrev_cplx` routine to produce optimized code.

Examples

The Bit Reverse block reorders the output of the C62xRadix-2 FFT in the model below to natural order.



The following code calculates the same FFT in the workspace. The output from this calculation, `y2`, is displayed side-by-side with the output from the model, c. The outputs match, showing that the Bit Reverse block reorders the Radix-2 FFT output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);

[y2, c]
   0.5000               0.5000
   0.4619 - 0.1913i     0.4619 - 0.1913i
   0.3536 - 0.3536i     0.3535 - 0.3535i
   0.1913 - 0.4619i     0.1913 - 0.4619i
        0 - 0.5000i          0 - 0.5000i
  -0.1913 - 0.4619i    -0.1913 - 0.4619i
  -0.3536 - 0.3536i    -0.3535 - 0.3535i
  -0.4619 - 0.1913i    -0.4619 - 0.1913i
  -0.5000              -0.5000
  -0.4619 + 0.1913i    -0.4619 + 0.1913i
  -0.3536 + 0.3536i    -0.3535 + 0.3535i
  -0.1913 + 0.4619i    -0.1913 + 0.4619i
        0 + 0.5000i          0 + 0.5000i
   0.1913 + 0.4619i     0.1913 + 0.4619i
   0.3536 + 0.3536i     0.3535 + 0.3535i
   0.4619 + 0.1913i     0.4619 + 0.1913i
```

See Also

C62xRadix-2 FFT, C62xRadix-2 IFFT

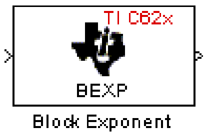
Purpose

Minimum number of extra sign bits in each input channel

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

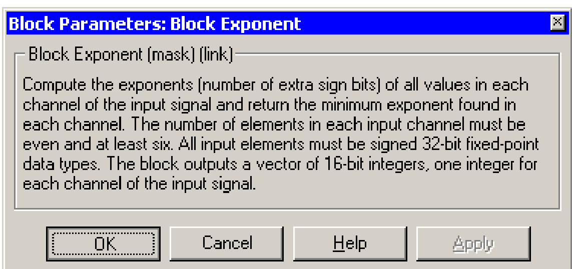


The Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be even and at least six. All input elements must be 32-bit signed fixed-point data types. The output is a vector of 16-bit integers — one integer for each channel of the input signal.

This block is useful for determining whether every sample in a channel is using extra sign bits. If so, you can scale your signal by the minimum number of extra sign bits to eliminate the common extra bits. This increases the representable precision and decreases the representable range of the signal.

The Block Exponent block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Block Exponent block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_bexp`. During code generation, this block calls the `DSP_bexp` routine given to produce optimized code.

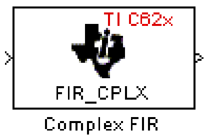
Purpose

Filter complex input signal using complex FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

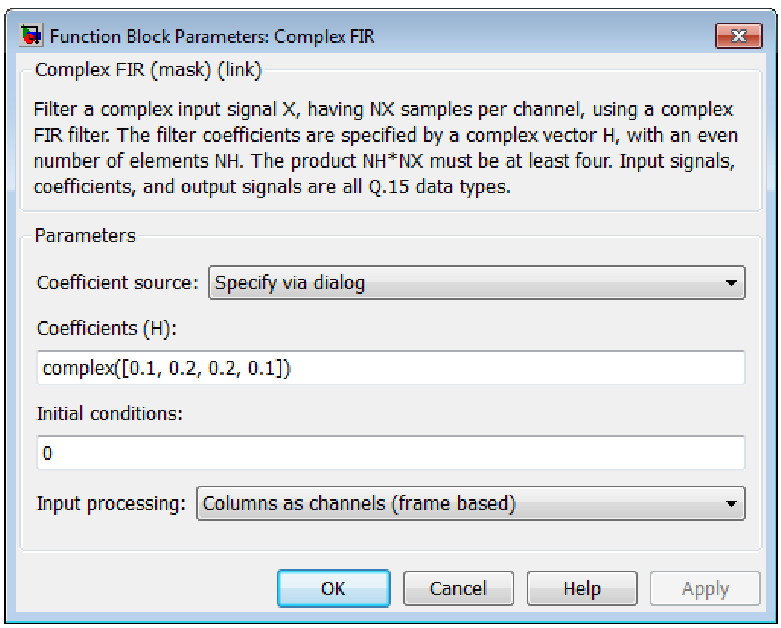


The Complex FIR block filters a complex input signal X using a complex FIR filter. This filter is implemented using a direct form structure.

The number of FIR filter coefficients, which are given as elements of the input vector H, must be even. The product of the number of elements of X and the number of elements of H must be at least four. Inputs, coefficients, and outputs are all Q.15 data types.

The Complex FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients (H)** parameter in the dialog
- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X.

Coefficients (H)

Designate the filter coefficients in vector format. There must be an even number of coefficients. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

You may enter real-valued initial conditions. Zero-valued imaginary parts will be assumed.

Input Processing

Process input signal as frames or samples

- `Columns as channels (frame based)` — Process the input signal as frames. Each frame contains a group of sequential data samples. To perform frame-based processing, you must have a DSP System Toolbox™ license.

- `Elements as channels (sample based)` — Process the input signal as individual data samples.
- `Inherited (this choice will be removed   see release notes)` — Use the frame status attribute of the input signal to determine whether to process the input as frames or samples.

  When you load an existing model in R2011a, the software sets this parameter to `Inherited (this choice will be removed - see release notes)`. Selecting this option allows you to continue working with your model until you upgrade. Upgrade your model using the `slupdate` function as soon as possible.

  Note For more information about this option, see "Changes to Frame-Based Processing"

Algorithm

In simulation, the Complex FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_cplx`. During code generation, this block calls the `DSP_fir_cplx` routine to produce optimized code.

See Also

C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

C62x Convert Floating-Point to Q.15

Purpose

Convert single-precision floating-point input signal to Q.15 fixed-point

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

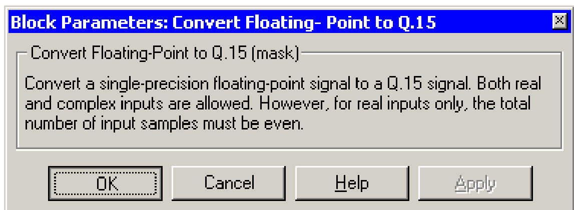
Description

Convert Floating-Point to Q.15

The Convert Floating-Point to Q.15 block converts a single-precision floating-point input signal to a Q.15 output signal. Input can be real or complex. For real inputs, the number of input samples must be even.

The Convert Floating-Point to Q.15 block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Convert Floating-Point to Q.15 block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fltoq15`. During code generation, this block calls the `DSP_fltoq15` routine to produce optimized code.

See Also

C62xConvert Q.15 to Floating Point

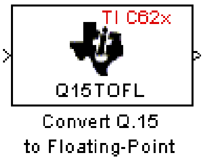
Purpose

Convert Q.15 fixed-point signal to single-precision floating-point

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

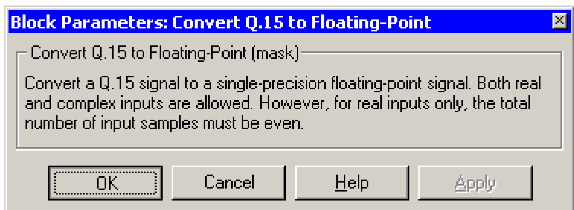
Description



The Convert Q.15 to Floating-Point block converts a Q.15 input signal to a single-precision floating-point output signal. Input can be real or complex. For real inputs, the number of input samples must be even.

The Convert Q.15 to Floating-Point block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Convert Q.15 to Floating-Point block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_q15tofl`. During code generation, this block calls the `DSP_q15tofl` routine to produce optimized code.

See Also

C62xConvert Floating-Point to Q.15

C62x FFT

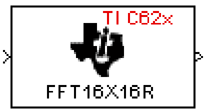
Purpose

Decimation-in-frequency forward FFT of complex input vector

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description



FFT

The FFT block computes the decimation-in-frequency forward FFT, with scaling between stages, of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 8 to 16,384, inclusive. The input must also be in natural (linear) order. The block outputs a complex signal in natural order. Inputs and outputs are signed 16-bit fixed-point data types.

The `fft16x16r` routine used by this block employs butterfly stages to perform the FFT. The number of butterfly stages used, S, depends on the input length L = 2^k. If k is even, then S = k/2. If k is odd, then S = (k+1)/2.

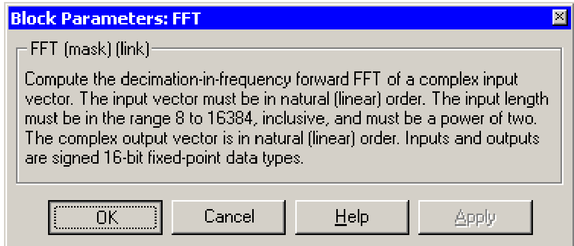
If k is even, then L is a power of two as well as a power of four, and this block performs all S stages with radix-4 butterflies to compute the output. If k is odd, then L is a power of two but not a power of four. In that case this block performs the first (S-1) stages with radix-4 butterflies, followed by a final stage using radix-2 butterflies.

To minimize noise, the FFT block also implements a divide-by-two scaling on the output of each stage except for the last. Therefore, to ensure that the gain of the block matches that of the theoretical FFT, the FFT block offsets the location of the binary point of the output data type by (S-1) bits to the right relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type minus (S-1).

$$OutputFractionalBits = InputFractionalBits - (S - 1)$$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



Algorithm

In simulation, the FFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fft16x16r`. During code generation, this block calls the `DSP_fft16x16r` routine to produce optimized code.

See Also

C62xRadix-2 FFT, C62xRadix-2 IFFT

C62x General Real FIR

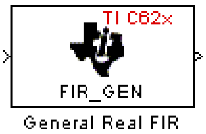
Purpose

Filter real input signal using real FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

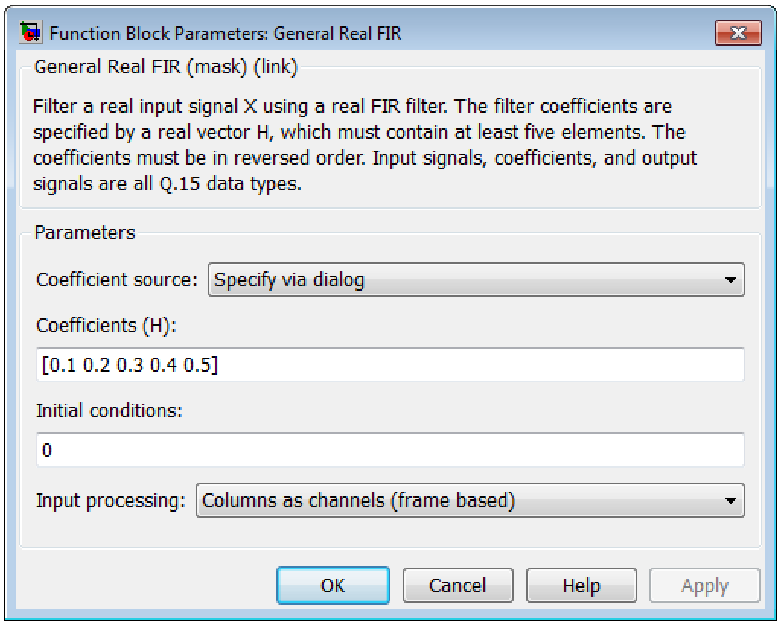


The General Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The filter coefficients are specified by a real vector H, which must contain at least five elements. The coefficients must be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The General Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients (H)** parameter in the dialog
- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)
Designate the filter coefficients in vector format. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions
If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

The initial conditions must be real.

Input Processing
Process input signal as frames or samples

- `Columns as channels (frame based)` — Process the input signal as frames. Each frame contains a group of sequential data samples. To perform frame-based processing, you must have a DSP System Toolbox license.
- `Elements as channels (sample based)` — Process the input signal as individual data samples.

- `Inherited (this choice will be removed   see release notes)` — Use the frame status attribute of the input signal to determine whether to process the input as frames or samples.

  When you load an existing model in R2011a, the software sets this parameter to `Inherited (this choice will be removed - see release notes)`. Selecting this option allows you to continue working with your model until you upgrade. Upgrade your model using the `slupdate` function as soon as possible.

  Note For more information about this option, see "Changes to Frame-Based Processing"

Algorithm

In simulation, the General Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_gen`. During code generation, this block calls the `DSP_fir_gen` routine to produce optimized code.

See Also

C62xComplex FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

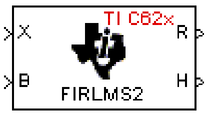
Purpose

LMS adaptive FIR filtering

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description



LMS Adaptive FIR

The LMS Adaptive FIR block performs least-mean-square (LMS) adaptive filtering. This filter is implemented using a direct form structure.

Note To implement a complete LMS algorithm, use this block in combination with the 5 other blocks shown in the "Examples" on page 5-390 section.

Note This block performs fixed-point computations using fixdt(1,16,15) and fixdt(1,32,30) data types. Because of this limitation, you may not be able to address numeric overflow and underflow problems with this block. As a result, this block is useful in a limited set of applications.

The following constraints apply to the inputs and outputs of this block:

- The scalar input X must be a Q.15 data type.
- The scalar input B must be a Q.15 data type.
- The scalar output R is a Q1.30 data type.
- The output $\bar{H}$ has length equal to the number of filter taps and is a Q.15 data type. The number of filter taps must be a positive, even integer.

This block performs LMS adaptive filtering according to the equations

$$e(n+1) = d(n+1) - [\bar{H}(n) \cdot \bar{X}(n+1)]$$

and

$$\bar{H}(n+1) = \bar{H}(n) + [\mu e(n+1) \cdot \bar{X}(n+1)],$$

where

- n designates the time step.
- $\bar{X}$ is a vector composed of the current and last nH–1 scalar inputs.
- d is the desired signal. The output R converges to d as the filter converges.
- $\bar{H}$ is a vector composed of the current set of filter taps.
- e is the error, or $d - [\bar{H}(n) \cdot \bar{X}(n+1)]$.
- μ is the step size.

For this block, the input B and the output R are defined by

$$B = \mu e(n+1)$$

and

$$R = \bar{H}(n) \cdot \bar{X}(n+1),$$

which combined with the first two equations, result in the following equations that this block follows:

$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)].$$

d and B must be produced externally to the LMS Adaptive FIR block. Refer to Examples below for a sample model that does this.

The LMS Adaptive FIR block supports discrete sample times and supports little-endian code generation only.

The rounding mode used is *floor*, and the saturation mode is *wrap*. All intermediate products have **s32Q30** data type. The update equation is as follows:

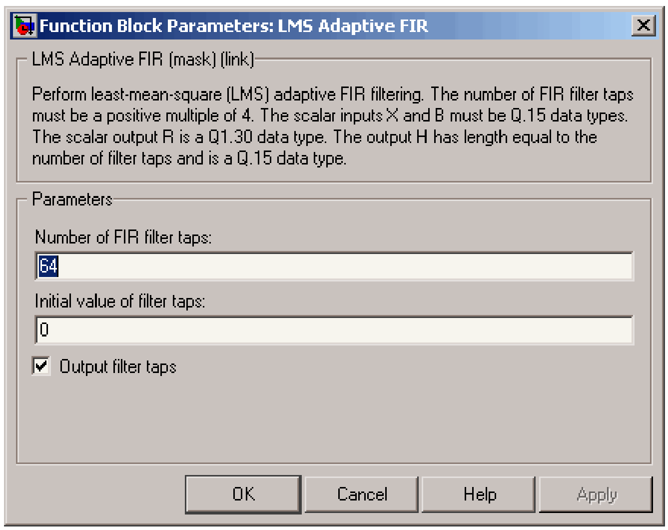
$$H_i = H_i + \mathtt{S16Q15}(\mathtt{S32Q30}(B) \times \mathtt{S32Q30}(X_i))$$

$$R = \sum_N (X_i \times H_i),$$

where N is the number of filter taps.

Note This block does not implement a leaky LMS algorithm, so comparison to the leakage factor of the LMS block of the DSP System Toolbox software is not appropriate.

Dialog Box



Number of FIR filter taps
: Designate the number of filter taps. The number of taps must be a positive, even integer.

Initial value of filter taps
: Enter the initial value of the filter taps.

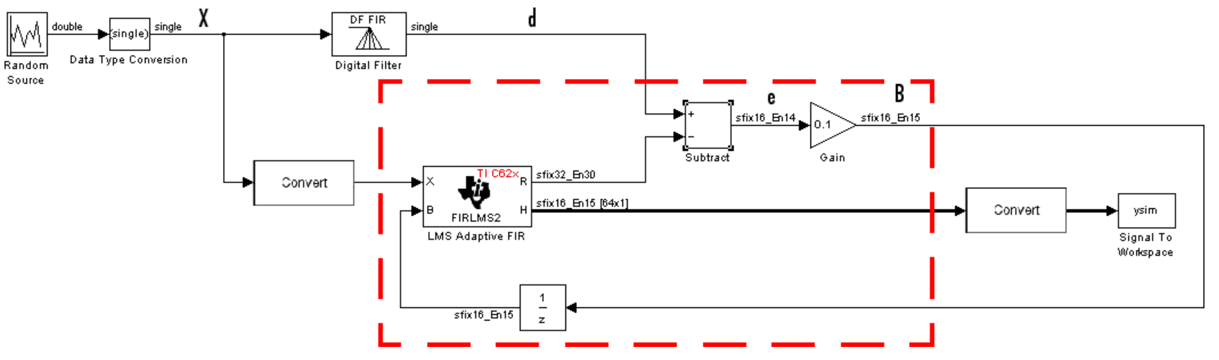
Output filter coefficients H?
: If you select this option, the filter taps are produced as output H. If not selected, H is suppressed.

Algorithm

In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_firlms2`. During code generation, this block calls the `DSP_firlms2` routine to produce optimized code.

Examples

The following model uses the LMS Adaptive FIR block.



The portion of the model enclosed by the dashed line produces the signal *B* and feeds it back into the LMS Adaptive FIR block. The inputs to this region are $\bar{X}$ and the desired signal *d*, and the output of this region is the vector of filter taps $\bar{H}$. Thus this region of the model acts as a canonical LMS adaptive filter. For example, compare this region to the `adaptfilt.lms` function in DSP System Toolbox software.

adaptfilt.lms performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

To use the LMS Adaptive FIR block you must create the input B in some way similar to the one shown here. You must also provide the signals $\bar{X}$ and d. This model simulates the desired signal d by feeding $\bar{X}$ into a digital filter block. You can simulate your desired signal in a similar way, or you may bring d in from the workspace with a From Workspace or codec block.

C62x Matrix Multiply

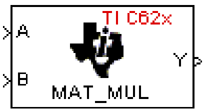
Purpose

Matrix multiply two input signals

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description



Matrix Multiply

The Matrix Multiply block multiplies two input matrices A and B. Inputs and outputs are real, 16-bit, signed fixed-point data types. This block wraps overflows when they occur.

The product of the two 16-bit inputs results in a 32-bit accumulator value. The Matrix Multiply block, however, only outputs 16 bits. You can choose to output the highest or second-highest 16 bits of the accumulator value.

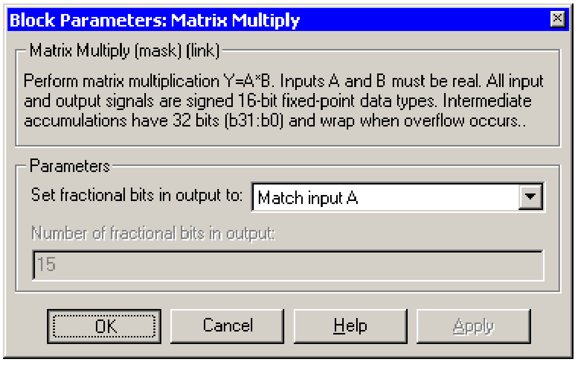
Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

	Input A	Input B	Accumulator Value
Total Bits	16	16	32
Fractional Bits	R	S	$R + S$

Therefore $R+S$ is the location of the binary point in the accumulator value. You can select 16 bits in relation to this fixed position of the accumulator binary point to give the desired number of fractional bits in the output (see Examples below). You can either require the output to have the same number of fractional bits as one of the two inputs, or you can specify the number of output fractional bits in the **Number of fractional bits in output** parameter.

The Matrix Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Choose which 16 bits to output from the list:

- `Match input A` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or *R* in the discussion above).
- `Match input B` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or *S* in the discussion above).
- `Match high bits of acc.  (b31:b16)` — Output the highest 16 bits of the accumulator value.
- `Match high bits of prod.  (b30:b15)` — Output the second-highest 16 bits of the accumulator value.
- `User-defined` — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select `User-defined` for **Set fractional bits in output to**.

Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_mul`. During code generation, this block calls the `DSP_mat_mul` routine to produce optimized code.

Examples

Example 1

Suppose A and B are both Q.15. The data type of the resulting accumulator value is therefore the 32-bit data type Q1.30 ($R + S = 30$). In the accumulator, bits 31:30 are the sign and integer bits, and bits 29:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
`Match input A`	Q.15	b30:b15
`Match input B`	Q.15	b30:b15
`Match high bits of acc.`	Q1.14	b31:b16
`Match high bits of prod.`	Q.15	b30:b15

Example 2

Suppose A is Q12.3 and B is Q10.5. The data type of the resulting accumulator value is therefore Q23.8 ($R + S = 8$). In the accumulator, bits 31:8 are the sign and integer bits, and bits 7:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
Match input A	Q12.3	b20:b5
Match input B	Q10.5	b18:b3
Match high bits of acc.	Q23.-8	b31:b16
Match high bits of prod.	Q22.-7	b30:b15

See Also

C62xVector Multiply

C62x Matrix Transpose

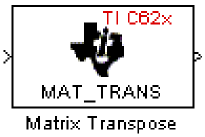
Purpose

Matrix transpose input signal

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

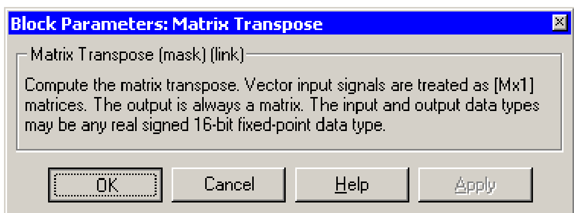


The Matrix Transpose block transposes an input matrix or vector. A 1-D input is treated as a column vector and is transposed to a row vector. Input and output signals are any real, 16-bit, signed fixed-point data type.

The Matrix Transpose block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Note If you use Target Function Library (TFL) technology with this block, the TI compiler generates processor and compiler-specific instructions that improve the performance of the generated code. For more information, consult“Introduction to Target Function Libraries”.

Dialog Box



Algorithm

In simulation, the Matrix Transpose block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mat_trans`. During code generation, this block calls the `DSP_mat_trans` routine to produce optimized code.

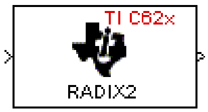
Purpose

Radix-2 decimation-in-frequency forward FFT of complex input vector

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description



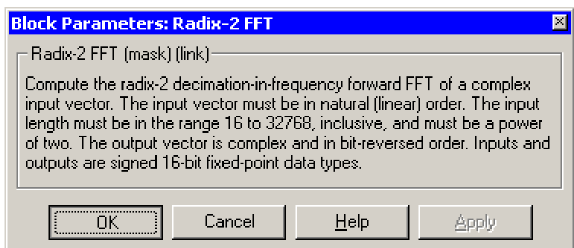
Radix-2 FFT

The Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types, and the output data type matches the input data type.

You can use the C62x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

The Radix-2 FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box

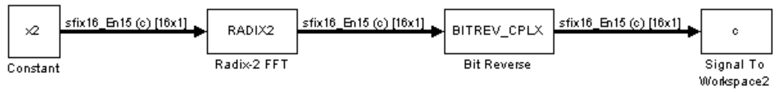


Algorithm

In simulation, the Radix-2 FFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

Examples

The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the C62x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.



The following code calculates the same FFT as the above model in the workspace. The output from this calculation, y2, is then displayed side-by-side with the output from the model, c. The outputs match, showing that the Bit Reverse block does reorder the Radix-2 FFT block output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);

[y2, c]
   0.5000              0.5000
   0.4619 - 0.1913i    0.4619 - 0.1913i
   0.3536 - 0.3536i    0.3535 - 0.3535i
   0.1913 - 0.4619i    0.1913 - 0.4619i
        0 - 0.5000i         0 - 0.5000i
  -0.1913 - 0.4619i   -0.1913 - 0.4619i
  -0.3536 - 0.3536i   -0.3535 - 0.3535i
  -0.4619 - 0.1913i   -0.4619 - 0.1913i
  -0.5000             -0.5000
  -0.4619 + 0.1913i   -0.4619 + 0.1913i
  -0.3536 + 0.3536i   -0.3535 + 0.3535i
  -0.1913 + 0.4619i   -0.1913 + 0.4619i
        0 + 0.5000i         0 + 0.5000i
   0.1913 + 0.4619i    0.1913 + 0.4619i
   0.3536 + 0.3536i    0.3535 + 0.3535i
   0.4619 + 0.1913i    0.4619 + 0.1913i
```

See Also

C62x Bit Reverse, C62x FFT, C62x Radix-2 IFFT

Purpose

Radix-2 inverse FFT of complex input vector

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

Radix-2 IFFT

The Radix-2 IFFT block computes the radix-2 inverse FFT of each channel of a complex input signal. This block uses a decimation-in-frequency forward FFT algorithm with butterfly weights modified to compute an inverse FFT. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types.

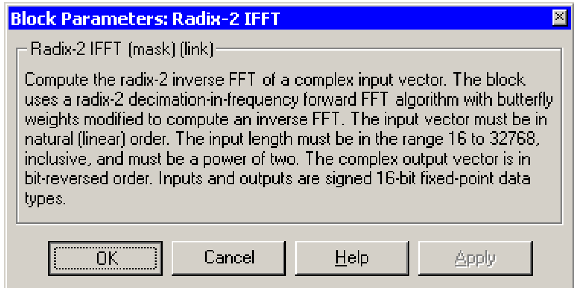
The `radix2` routine used by this block employs a radix-2 FFT of length L=2^k. So that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by k bits to the left relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type plus k.

$OutputFractionalBits = InputFractionalBits + (k)$

You can use the C62x Bit Reverse block to reorder the output of the Radix-2 IFFT block to natural order.

The Radix-2 IFFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



Algorithm

In simulation, the Radix-2 IFFT block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

See Also

C62x Bit Reverse, C62x FFT, C62x Radix-2 FFT

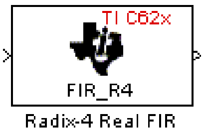
Purpose

Filter real input signal using real FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

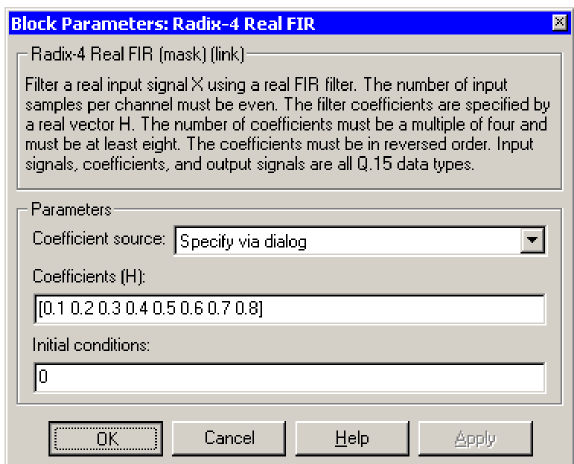


The Radix-4 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be even. The filter coefficients are specified by a real vector, H. The number of filter coefficients must be a multiple of four and must be at least eight. The coefficients must also be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-4 Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients** parameter in the dialog

- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-4 Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_r4`. During code generation, this block calls the `DSP_fir_r4` routine to produce optimized code.

See Also

C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-8 Real FIR, C62xSymmetric Real FIR

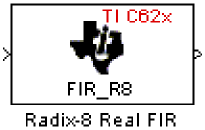
Purpose

Filter real input signal using real FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

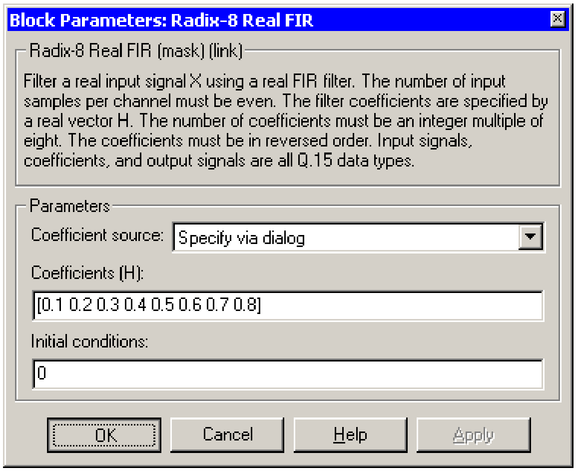


The Radix-8 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be even. The filter coefficients are specified by a real vector, H. The number of coefficients must be an integer multiple of eight. The coefficients must be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-8 Real FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients** parameter in the dialog

- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)
Designate the filter coefficients in vector format. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions
If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-8 Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_r8`. During code generation, this block calls the `DSP_fir_r8` routine to produce optimized code.

See Also

C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xSymmetric Real FIR

Purpose

Filter real input signal using lattice filter

Library

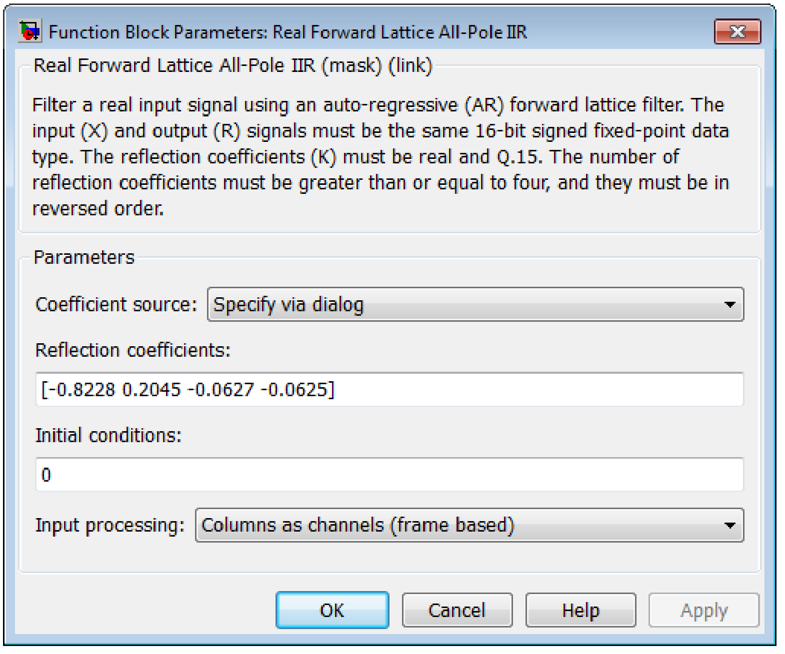
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

The Real Forward Lattice All-Pole IIR block filters a real input signal using an autoregressive forward lattice filter. The input and output signals must be the same 16-bit signed fixed-point data type. The reflection coefficients must be real and Q.15. The number of reflection coefficients must be greater than or equal to four, and they must be in reversed order. Use an even number of reflection coefficients to maximize the speed of your generated code.

The Real Forward Lattice All-Pole IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Reflection coefficients** parameter in the dialog
- `Input port` — Accept the coefficients from port K

Reflection coefficients

Designate the reflection coefficients of the filter in vector format. The number of coefficients must be greater than or equal to four, and they must be in reverse order. Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select `Specify via dialog` for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If your block initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length (number of elements) of this vector must be the same as the number of reflection coefficients in your filter.
- Different across channels, enter a matrix containing all initial conditions. The number of rows (initial conditions for one channel) of this matrix must be the same as the number of reflection coefficients, and the number of columns of this matrix must be equal to the number of channels.

Input Processing

Process input signal as frames or samples

- `Columns as channels (frame based)` — Process the input signal as frames. Each frame contains a group of sequential data samples. To perform frame-based processing, you must have a DSP System Toolbox license.

- `Elements as channels (sample based)` — Process the input signal as individual data samples.
- `Inherited (this choice will be removed   see release notes)` — Use the frame status attribute of the input signal to determine whether to process the input as frames or samples.

  When you load an existing model in R2011a, the software sets this parameter to `Inherited (this choice will be removed - see release notes)`. Selecting this option allows you to continue working with your model until you upgrade. Upgrade your model using the `slupdate` function as soon as possible.

Note For more information about this option, see "Changes to Frame-Based Processing"

Algorithm

In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_iirlat`. During code generation, this block calls the `DSP_iirlat` routine to produce optimized code.

See Also

C62xReal IIR

C62x Real IIR

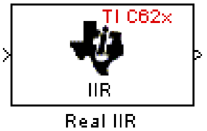
Purpose

Filter real input signal using IIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description



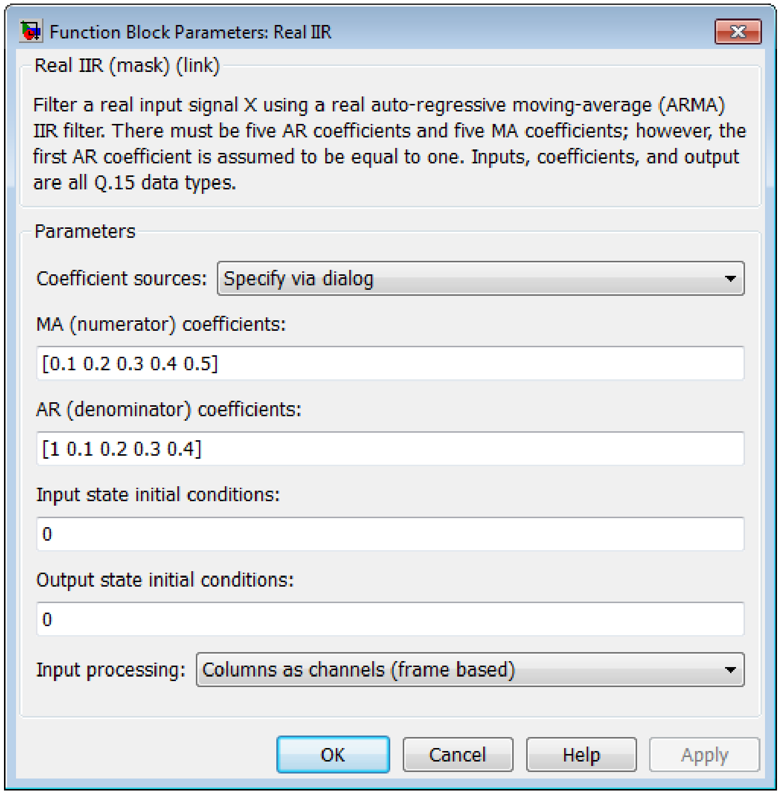
Real IIR

The Real IIR block filters a real input signal X using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure.

There must be five AR coefficients and five MA coefficients. The first AR coefficient is always assumed to be one. Inputs, coefficients, and output are Q.15 data types.

The Real IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient sources

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **MA (numerator) coefficients** and **AR (denominator) coefficients** parameters in the dialog
- `Input ports` — Accept the coefficients from ports MA and AR

MA (numerator) coefficients

Designate the moving-average coefficients of the filter in vector format. There must be five MA coefficients. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

AR (denominator) coefficients
Designate the autoregressive coefficients of the filter in vector format. There must be five AR coefficients, however the first AR coefficient is assumed to be equal to one. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

Input state initial conditions
If the input state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the input state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all input state initial conditions. This matrix must have four rows.

Output state initial conditions
If the output state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the output state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all output state initial conditions. This matrix must have four rows.

Input Processing
Process input signal as frames or samples

- `Columns as channels (frame based)` — Process the input signal as frames. Each frame contains a group of sequential data samples. To perform frame-based processing, you must have a DSP System Toolbox license.

- `Elements as channels (sample based)` — Process the input signal as individual data samples.
- `Inherited (this choice will be removed   see release notes)` — Use the frame status attribute of the input signal to determine whether to process the input as frames or samples.

  When you load an existing model in R2011a, the software sets this parameter to `Inherited (this choice will be removed - see release notes)`. Selecting this option allows you to continue working with your model until you upgrade. Upgrade your model using the `slupdate` function as soon as possible.

Note For more information about this option, see "Changes to Frame-Based Processing"

Algorithm

In simulation, the Real IIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_iir`. During code generation, this block calls the `DSP_iir` routine to produce optimized code.

See Also

C62xReal Forward Lattice All-Pole IIR

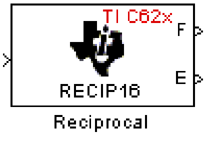
Purpose

Fraction and exponent portions of reciprocal of real input signal

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

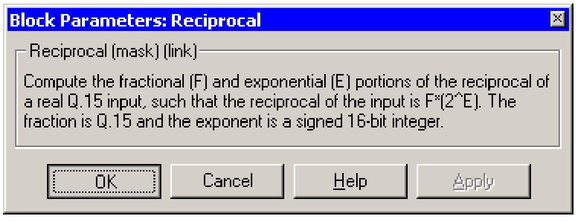
Description



The Reciprocal block computes the fractional (F) and exponential (E) portions of the reciprocal of a real Q.15 input, such that the reciprocal of the input is $F*(2^E)$. The fraction is Q.15 and the exponent is a 16-bit signed integer.

The Reciprocal block supports both continuous and discrete sample times. This block also supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Reciprocal block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_recip16`. During code generation, this block calls the `DSP_recip16` routine to produce optimized code.

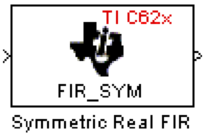
Purpose

Filter real input signal using FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description



The Symmetric Real FIR block filters a real input signal using a symmetric real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be even. The filter coefficients are specified by a real vector H, which must be symmetric about its middle element. The number of coefficients must be of the form 16k + 1, where k is a positive integer. This block wraps overflows that occur. The input, coefficients, and output are 16-bit signed fixed-point data types.

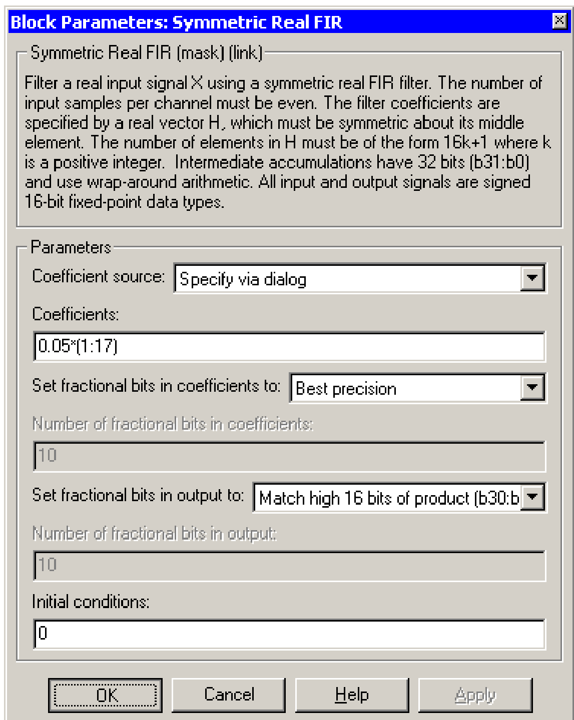
Intermediate multiplies and accumulates performed by this filter result in a 32-bit accumulator value. However, the Symmetric Real FIR block only outputs 16 bits. You can choose to output 16 bits of the accumulator value in one of the following ways.

`Match input x`	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the input
`Match coefficients h`	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the coefficients
`Match high 16 bits of acc.`	Output bits 31 - 16 of the accumulator value

`Match high 16 bits of prod.`	Output bits 30 - 15 of the accumulator value
`User-defined`	Output 16 bits of the accumulator value such that the output has the number of fractional bits specified in the **Number of fractional bits in output** parameter

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients** parameter in the dialog
- `Input port` — Accept the coefficients from port H

Coefficients

Enter the coefficients in vector format. This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter. This parameter is tunable in simulation.

Set fractional bits in coefficients to

Specify the number of fractional bits in the filter coefficients:

- `Match input X` — Sets the coefficients to have the same number of fractional bits as the input
- `Best precision` — Sets the number of fractional bits of the coefficients such that the coefficients are represented to the best precision possible
- `User-defined` — Sets the number of fractional bits in the coefficients with the **Number of fractional bits in coefficients** parameter

This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter.

Number of fractional bits in coefficients

Specify the number of bits to the right of the binary point in the filter coefficients. This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter, and is only enabled if `User-defined` is specified for the **Set fractional bits in coefficients to** parameter.

Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Select which 16 bits to output:

- `Match input X` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input X
- `Match coefficients H` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in coefficients H
- `Match high bits of acc. (b31:b16)` — Output the highest 16 bits of the accumulator value
- `Match high bits of prod. (b30:b15)` — Output the second-highest 16 bits of the accumulator value
- `User-defined` — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter

See Matrix Multiply "Examples" on page 5-394 for demonstrations of these selections.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is only enabled if `User-defined` is selected for the **Set fractional bits in output to** parameter.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less

than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Algorithm

In simulation, the Symmetric Real FIR block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_fir_sym`. During code generation, this block calls the `DSP_fir_sym` routine to produce optimized code.

See Also

C62xComplex FIR, C62xGeneral Real FIR, C62xRadix-4 Real FIR, C62xRadix-8 Real FIR

C62x Vector Dot Product

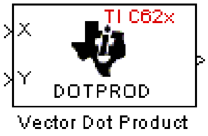
Purpose

Vector dot product of real input signals

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

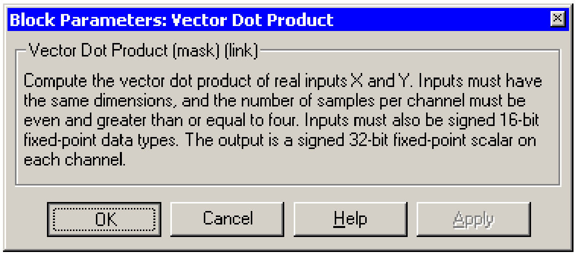
Description



The Vector Dot Product block computes the vector dot product of two real input vectors, X and Y. The input vectors must have the same dimensions and must be signed 16-bit fixed-point data types. The number of samples per channel of the inputs must be even and greater than or equal to four. The output is a signed 32-bit fixed-point scalar on each channel, and the number of fractional bits of the output is equal to the sum of the number of fractional bits of the inputs.

The Vector Dot Product block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Dot Product block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_dotprod`. During code generation, this block calls the `DSP_dotprod` routine to produce optimized code.

Purpose

Zero-based index of maximum value element in each input signal channel

Library

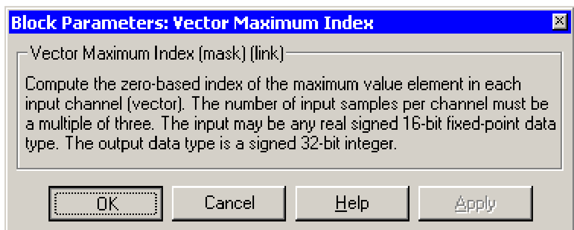
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

The Vector Maximum Index block computes the zero-based index of the maximum value element in each channel (vector) of the input signal. The input may be any real, 16-bit, signed fixed-point data type, and the number of samples per input channel must be an integer multiple of three. The output data type is a 32-bit signed integer.

The Vector Maximum Index block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Maximum Index block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_maxidx`. During code generation, this block calls the `DSP_maxidx` routine to produce optimized code.

C62x Vector Maximum Value

Purpose

Maximum value for each input signal channel

Library

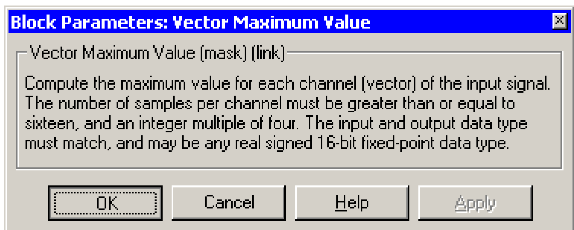
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

The Vector Maximum Value block returns the maximum value in each channel (vector) of the input signal. The input can be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of four and must be at least 16. The output data type matches the input data type.

The Vector Maximum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Maximum Value block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_maxval`. During code generation, this block calls the `DSP_maxval` routine to produce optimized code.

See Also

C62xVector Minimum Value

Purpose

Minimum value for each input signal channel

Library

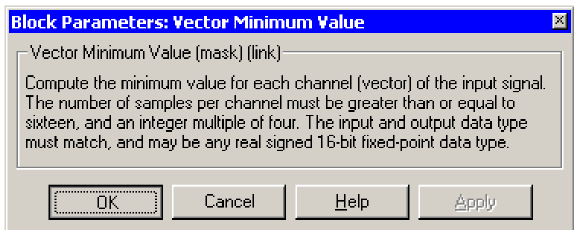
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

Description

The Vector Minimum Value block returns the minimum value in each channel of the input signal. The input may be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of four and must be at least 16. The output data type matches the input data type.

The Vector Minimum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Minimum Value block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_minval`. During code generation, this block calls the `DSP_minval` routine to produce optimized code.

See Also

C62xVector Maximum Value

C62x Vector Multiply

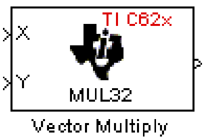
Purpose

Element-wise multiplication on inputs

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

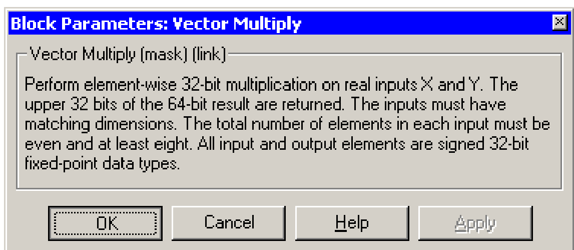
Description



The Vector Multiply block performs element-wise 32-bit multiplication of two inputs X and Y. The total number of elements in each input must be even and at least eight, and the inputs must have matching dimensions. The upper 32 bits of the 64-bit accumulator result are returned. All input and output elements are 32-bit signed fixed-point data types.

The Vector Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Multiply block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_mul32`. During code generation, this block calls the `DSP_mul32` routine to produce optimized code.

See Also

C62xMatrix Multiply

C62x Vector Negate

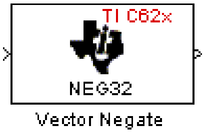
Purpose

Negate each input signal element

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

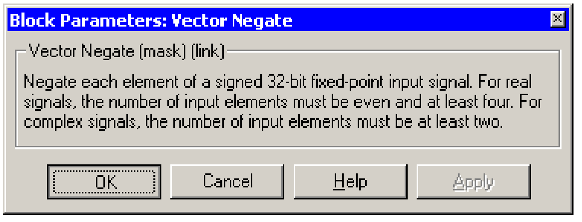
Description



The Vector Negate block negates each element of a 32-bit signed fixed-point input signal. For real signals, the number of input elements must be even and at least four. For complex signals, the number of input elements must be at least two. The output is the same data type as the input.

The Vector Negate block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Negate block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_neg32`. During code generation, this block calls the `DSP_neg32` routine to produce optimized code.

C62x Vector Sum of Squares

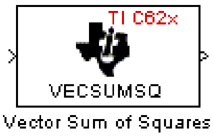
Purpose

Sum of squares over each real input channel

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

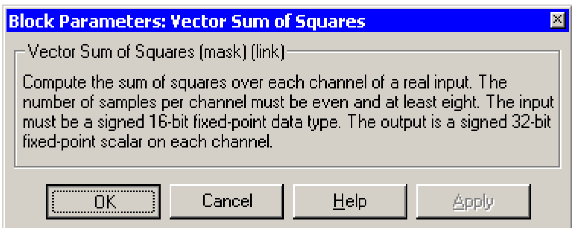
Description



The Vector Sum of Squares block computes the sum of squares over each channel of a real input. The number of samples per input channel must be even and at least eight, and the input must be a 16-bit signed fixed-point data type. The output is a 32-bit signed fixed-point scalar on each channel. The number of fractional bits of the output is twice the number of fractional bits of the input.

The Vector Sum of Squares block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



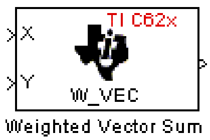
Algorithm

In simulation, the Vector Sum of Squares block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_vecsumsq`. During code generation, this block calls the `DSP_vecsumsq` routine to produce optimized code.

Purpose Weighted sum of input vectors

Library Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C62x DSP Library

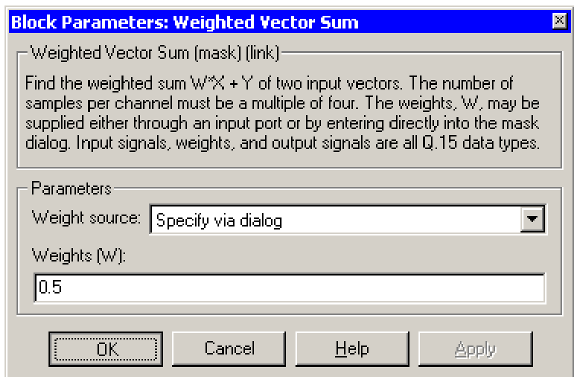
Description



The Weighted Vector Sum block computes the weighted sum of two inputs, X and Y, according to (W*X)+Y. Inputs may be vectors or frame-based matrices. The number of samples per channel must be a multiple of four. Inputs, weights, and output are Q.15 data types, and weights must be in the range -1 < W < 1.

The Weighted Vector Sum block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Weight source

Specify the source of the weights:

- `Specify via dialog` — Enter the weights in the **Weights (W)** parameter in the dialog
- `Input port` — Accept the weights from port W

Weights (W)

This parameter is visible only when `Specify via dialog` is specified for the **Weight source** parameter. This parameter is tunable in simulation. When the weights are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be a multiple of four.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be a multiple of four, and the number of columns of this matrix must be equal to the number of channels.

Weights must be in the range -1 < W < 1.

Algorithm

In simulation, the Weighted Vector Sum block is equivalent to the TMS320C62x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

C6416 DSK ADC

Purpose

Digitized output from codec to processor

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6416 DSK

Description

ADC

Use the C6416 DSK ADC (analog-to-digital converter) block to capture and digitize analog signals from the analog input jacks on the board. Placing an C6416 DSK ADC block in your Simulink block diagram lets you use the AIC23 coder-decoder module (codec) on the C6416 DSK to convert an analog input signal to a digital signal for the digital signal processor.

Most of the configuration options in the block affect the codec. However, the **Output data type**, **Samples per frame**, and **Scaling** options relate to the model you are using in Simulink software, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each option listed with the C6416 DSK hardware affected.

Option	Affected Hardware
ADC Source	Codec
Mic	Codec
Output data type	TMS320C6416 digital signal processor
Samples per frame	Direct memory access module
Sample Rate	Codec
Scaling	TMS320C6416 digital signal processor
Word Length	Codec

You can select one of two input sources from the **ADC source** list:

- **Line In** — the codec accepts input from the line in connector (LINE IN) on the board's mounting bracket.

- **Mic** — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

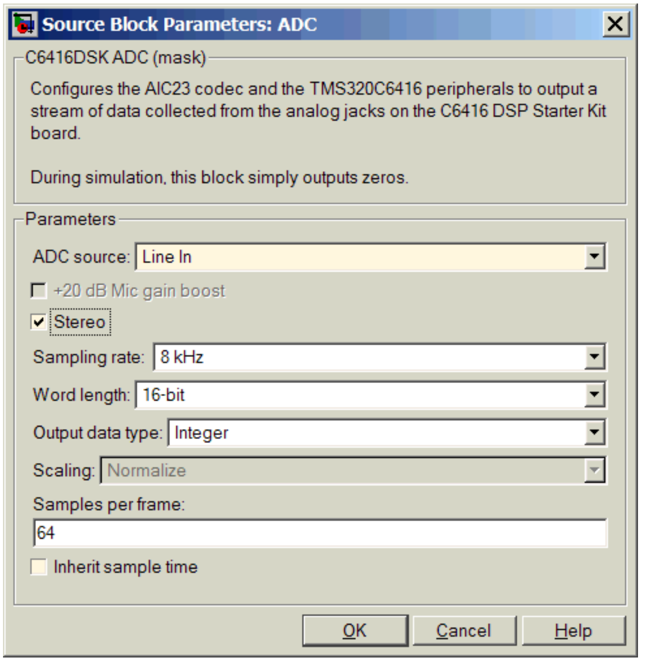
Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels on input and output.

The block uses frame-based processing of inputs, buffering the input data into frames at the specified samples per frame rate. In Simulink software, the block puts monaural data into an N-element column vector. Stereo data input forms an N-by-2 matrix with N data values and two stereo channels (left and right).

When the samples per frame setting is more than one, each frame of data is either the N-element vector (monaural input) or N-by-2 matrix (stereo input). For monaural input, the elements in each frame form the column vector of input audio data. In the stereo format, the frame is the matrix of audio data represented by the matrix rows and columns — the rows are the audio data samples and the columns are the left and right audio channels.

When you select `Mic` for **ADC source**, you can select the **+20 dB Mic gain boost** check box to add 20 dB to the microphone input signal before the codec digitizes the signal.

Dialog Box



ADC source

The input source to the codec. `Line In` is the default. Selecting `Mic` enables the **+20 dB Mic gain boost** option.

+20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is `Mic`. Gain is applied before analog-to-digital conversion.

Stereo

Indicates whether the input audio data is in monaural or stereo format. Select the check box to enable stereo input. Clear the check box when you input monaural data. By default, stereo is enabled. Monaural data comes from the right channel.

Sample rate

Sets the sample rate for the data output by the codec. Options are `8`, `32`, `44.1`, `48`, and `96 kHz`, with a default of `8 kHz`.

Word length
Sets the resolution with which the ADC samples the analog input. Increasing the word length increases the accuracy of the data in each sample. If your model also contains a DAC block, set its word length match that of the ADC block.

Output data type
Selects the word length and shape of the data from the codec. By default, `double` is selected. Options are `Double`, `Single`, and `Integer`. To process single and double data types, the block uses emulated floating-point instructions on the C6416 processor.

Scaling
Selects whether the codec data is unmodified, or normalized to the output range to ±1.0, based on the codec data format. Select either `Normalize` or `Integer` from the list. `Normalize` is the default setting.

Samples per frame
Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal the block 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. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 8000 samples per second, and you select 32 samples per frame, the frame rate is 250 frames per second. The throughput remains the same at 8000 samples per second.

Inherit sample time
Selects whether the block inherits the sample time from the model base rate or Simulink base rate as determined 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

C6416 DSK DAC

C6416 DSK DAC

Purpose

Use codec to convert digital input to analog output

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6416 DSK

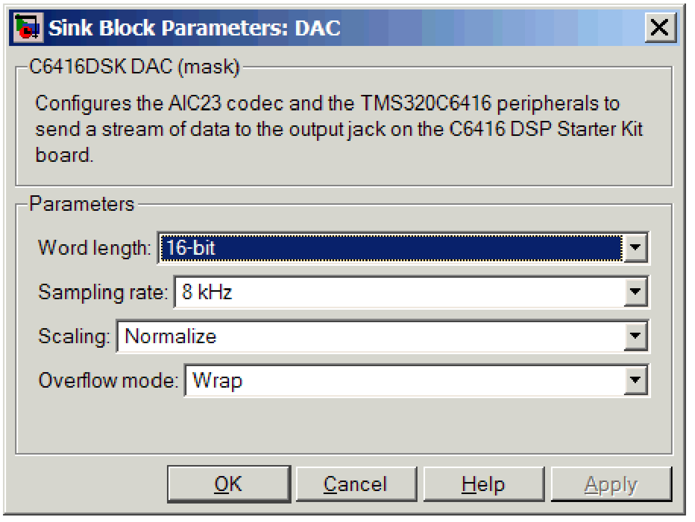
Description

Adding the C6416 DSK DAC (digital-to-analog converter) block to your Simulink model lets you output an analog signal to the LINE OUT connection on the C6416 DSK board. When you add the C6416 DSK DAC block, the digital signal received by the codec is converted to an analog signal and sent to the output jack.

Only the **Word length** option in the block affects the codec. The other options relate to the model you are using in Simulink software and the signal processor on the board. Refer to the following table for information.

Option	Affected Hardware
Overflow mode	TMS320C6416 Digital Signal Processor
Scaling	TMS320C6416 Digital Signal Processor
Word length	Codec

Dialog Box



Word length

Sets the DAC to interpret the input data word length. Without this setting, the DAC cannot convert the digital data to analog correctly. The value defaults to 16 bits, with options of 20, 24, and 32 bits. The word length you set here should always match the ADC setting.

Sampling rate

Sets the sampling rate for the block output to the output ports on the target. Select from the list of available rates.

Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range ±1.0. Matching the setting for the C6416 DSK ADC block is usually appropriate here.

Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter. You can choose `Wrap` or `Saturate` to handle the result of an overflow in an operation. If efficient operation matters, `Wrap` is the more efficient mode.

See Also

C6416 DSK ADC

C6416 DSK DIP Switch

Purpose

Simulate or read DIP switches

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6416 DSK

Description

Added to your model, this block behaves differently in simulation than in code generation and targeting.

In Simulation — the options **Switch 0**, **Switch 1**, **Switch 2,** and **Switch 3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6416 DSK. Each option turns the associated DIP switch on when you select it. The switches are independent of one another.

By defining the switches to represent actions on your target, DIP switches let you modify the operation of your process by reconfiguring the switch settings.

Use the **Data type** to specify whether the DIP switch options output an integer or a logical string of bits to represent the status of the switches. The table that follows presents all the option setting combinations with the result of your **Data type** selection.

Option Settings to Simulate the User DIP Switches on the C6416 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Cleared	Cleared	Cleared	0000	0
Selected	Cleared	Cleared	Cleared	0001	1
Cleared	Selected	Cleared	Cleared	0010	2
Selected	Selected	Cleared	Cleared	0011	3
Cleared	Cleared	Selected	Cleared	0100	4
Selected	Cleared	Selected	Cleared	0101	5

Option Settings to Simulate the User DIP Switches on the C6416 DSK (Continued)

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Selected	Selected	Cleared	0110	6
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9
Cleared	Selected	Cleared	Selected	1010	10
Selected	Selected	Cleared	Selected	1011	11
Cleared	Cleared	Selected	Selected	1100	12
Selected	Cleared	Selected	Selected	1101	13
Cleared	Selected	Selected	Selected	1110	14
Selected	Selected	Selected	Selected	1111	15

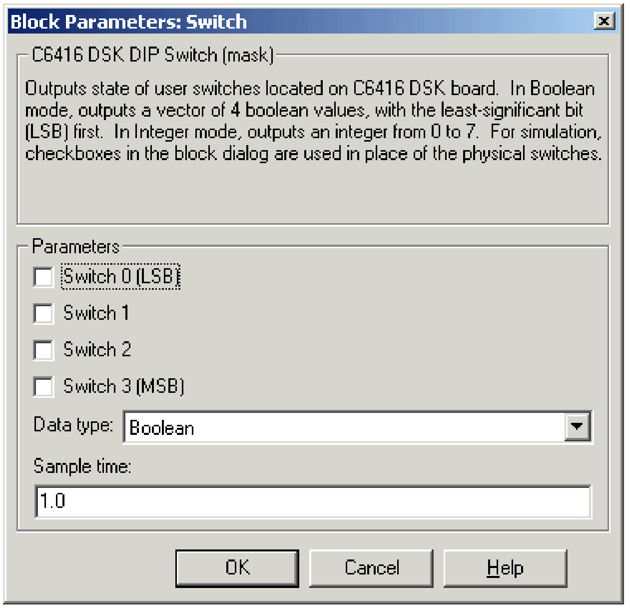
Selecting the `Integer` data type results in the switch settings generating integers in the range from 0 to 15 (uint8), corresponding to converting the string of individual switch settings to a decimal value. In the `Boolean` data type, the output string presents the separate switch setting for each switch, with the **Switch 0** status represented by the least significant bit (LSB) and the status of **Switch 3** represented by the most significant bit (MSB).

In Code generation and targeting — the code generated by the block reads the physical switch settings of the user switches on the board and reports them as shown in the table above. Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running your application, the block code ignores the settings for **Switch 0**, **Switch 1**, **Switch 2** and **Switch 3** in favor of reading the hardware switch settings. When the block reads the DIP switches, it reports the results as either a Boolean string or an integer value as the following table shows.

Output Values From The User DIP Switches on the C6416 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2
On	On	Off	Off	0011	3
Off	Off	On	Off	0100	4
On	Off	On	Off	0101	5
Off	On	On	Off	0110	6
On	On	On	Off	0111	7
Off	Off	Off	On	1000	8
On	Off	Off	On	1001	9
Off	On	Off	On	1010	10
On	On	Off	On	1011	11
Off	Off	On	On	1100	12
On	Off	On	On	1101	13
Off	On	On	On	1110	14
On	On	On	On	1111	15

Dialog Box



Switch 0
Simulate the status of the user-defined DIP switch on the board.

Switch 1
Simulate the status of the user-defined DIP switch on the board.

Switch 2
Simulate the status of the user-defined DIP switch on the board.

Switch 3
Simulate the status of the user-defined DIP switch on the board.

Data type
Determines how the block reports the status of the user-defined DIP switches. `Boolean` is the default, indicating that the output is a vector of four logical values.

Each vector element represents the status of one DIP switch; the first is **Switch 0** and the fourth is **Switch 3**. The data type

`Integer` converts the logical string to an equivalent unsigned 8-bit (`uint8`) value. For example, when the logical string generated by the switches is 0101, the conversion yields 5 — the MSB is 0 and the LSB is 1.

Sample time

Specifies the time between samples of the signal. This value defaults to 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

Purpose

Control LEDs

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6416 DSK

Description

Adding the C6416 DSK LED block to your Simulink block diagram lets you trigger the user light emitting diodes (LED) on the C6416 DSK. To use the block, send a nonzero real scalar to the block. The C6416 DSK LED block controls all four User LEDs located on the C6416 DSK.

When you add this block to a model, and send an integer to the block input, the block sets the LED state based on the input value it receives:

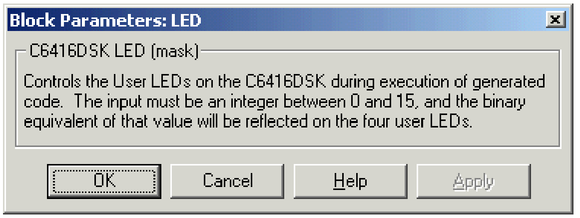
- When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000
- When the block receives a nonzero input value, the specified LEDs are turned on (enabled), 0001 to 1111

To activate the block, send it an integer in the range 0 to 15. Vectors do not work to activate LEDs; nor do complex numbers as scalars or vectors.

For example, sending the value 6 turns on the diodes to show 0110 (off/on/on/off). 13 turns on the diodes to show 1101.

All LEDs maintain their state until the C6416 DSK LED block receives an input value that changes the state. Enabled LEDs stay on until the block receives an input value that turns the LEDs off; disabled LEDs stay off until turned on. Resetting the C6416 DSK turns off all User LEDs. When you start an application, the LEDs are turned off by default.

Dialog Box



This dialog does not have any user-selectable options.

Purpose

Reset to initial conditions

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6416 DSK

Description

Double-clicking this block in a Simulink model window resets the C6416 DSK that is running the executable code built from the model. When you double-click the C6416 DSK Reset block, the block runs the software reset function provided by CCS IDE that resets the processor on your C6416 DSK. Applications running on the board stop and the signal processor returns to the initial conditions you defined.

Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library, it resets your C6416 DSK. In other words, any time you double-click a C6416 DSK Reset block, you reset your C6416 DSK.

Dialog Box

This block does not have settable options and does not provide a user interface dialog.

Purpose

Configure AIC23 audio codec to capture audio stream from LINE-IN or MIC

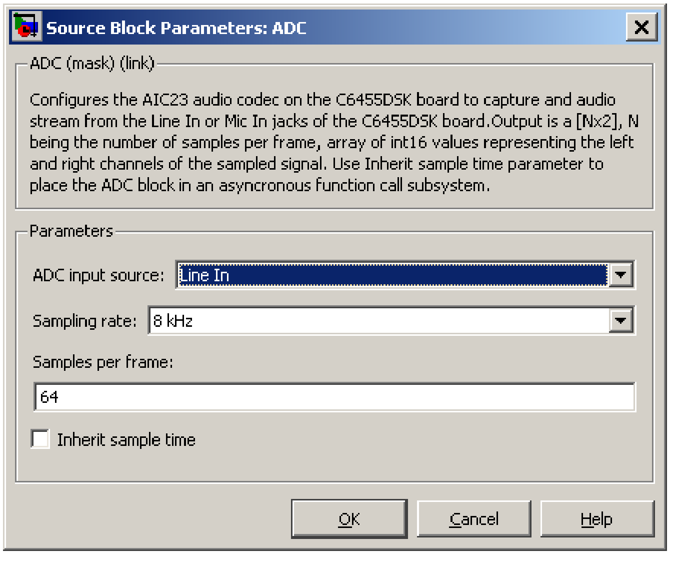
Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

Description

This block uses the AIC23 audio codec on the C6455 DSK/EVM board to capture an analog audio stream from the **Line In** or **Mic** jacks and generate a digital frame-based output. Output is a [Nx2] array of int16 values representing the left and right channels of the sampled signal, where N is the number of samples per frame. Use the **Inherit sample time** parameter to place the ADC block in an asynchronous function call subsystem.

Dialog Box



ADC input source
Select **Line In** or **Mic In** as the input source.

Sampling Rate

Set the sampling rate of the analog-to-digital converter. Increasing the frequency increases the accuracy of the sampling data over time.

Samples per frame

Set the number of samples the block 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. For example, if **Sampling Rate** is 8 kHz, and **Samples per frame** is 32, the 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 Simulink base rate as determined 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

DM6437 EVM DAC

C6455 DSK/EVM DAC

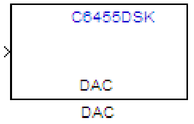
Purpose

Configure AIC23 codec to convert digital signal to audio output on LINE OUT and HP OUT

Library

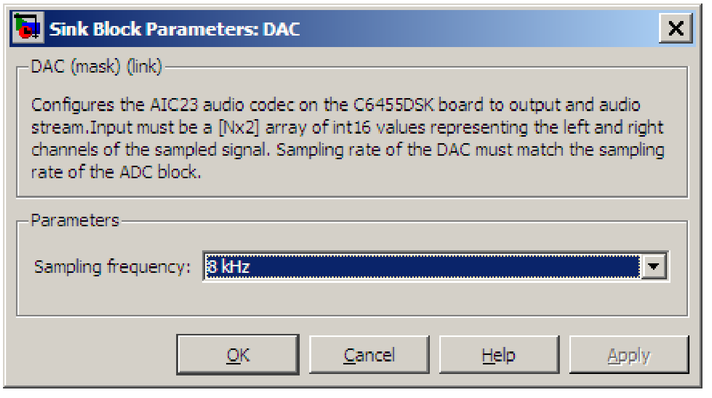
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

Description



Configure the AIC23 stereo codec on the C6455 EVM board to convert a digital signal to an analog audio stream on the LINE OUT and HP OUT output jacks. The digital signal input must be an [Nx2] array of int16 values. Column 1 of the array is the left channel and column 2 is the right channel of the sampled signal. The sampling rate of the DAC output must match the sampling rate of the digital signal from the ADC.

Dialog Box



Sampling Frequency

Set the sampling rate of the digital-to-analog converter. The rate defaults to 8 kHz. Options range up to 96 kHz.

See Also

C6455 DSK/EVM ADC

C6455 DSK/EVM DIP

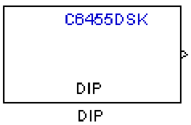
Purpose

Output state of user-selected DIP switch as Boolean

Library

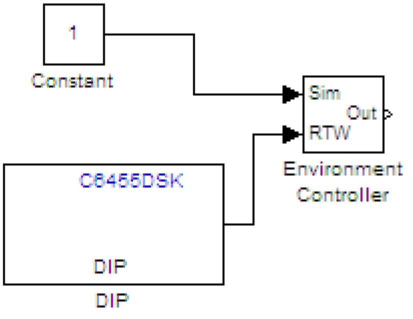
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

Description

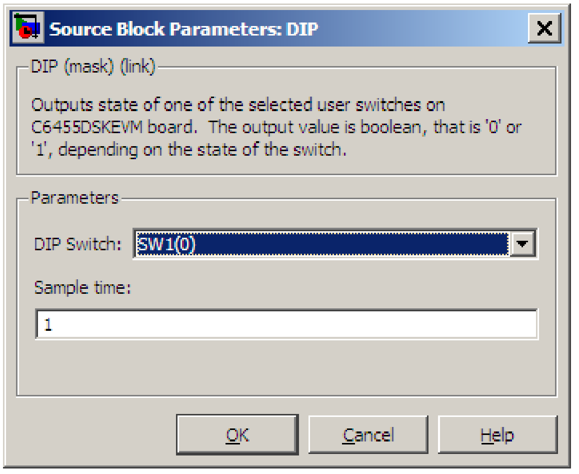


Outputs a Boolean that gives the state of a user-selected DIP switch from the SW1 bank of switches on the C6455 DSK/EVM board. Boolean 0 means the switch is open, and Boolean 1 means it is closed. Use multiple blocks to output the state of multiple DIP switches.

For simulations, you may want to use the C6455 DSK/EVM DIP block with a Constant block and an Environment Controller block, both from the Simulink block libraries.



Dialog Box



DIP Switch

Select the switch, 0 through 3, from the SW1 bank of switches.

Sample Time

Specifies the time between samples of the signal in seconds. This value defaults to 1 second between samples.

C6455 DSK/EVM LED

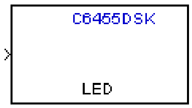
Purpose

Apply Boolean input to user-selected LED

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

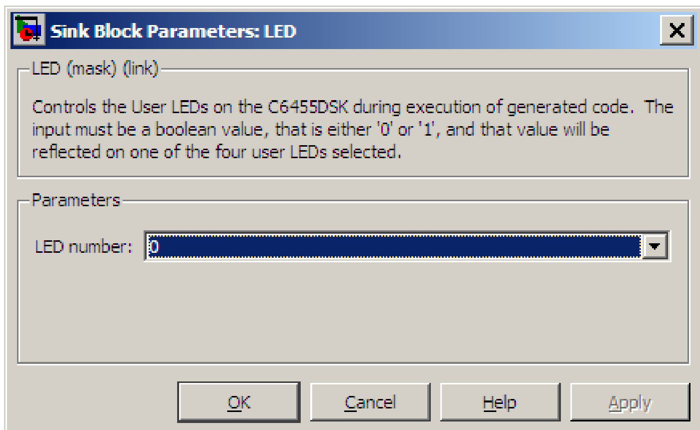
Description



LED

This block controls an individual LED among the User LEDs on the C6455 DSK/EVM during execution of generated code. The block input accepts Boolean values, 0 (off) or 1 (on). Use multiple blocks to control multiple LEDs.

Dialog Box



LED number

Specify the number of the User LED that the Boolean input controls.

C6455 SRIO Config

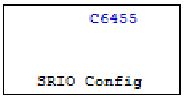
Purpose

Configure generated code for serial RapidI/O peripheral

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

Description



SRIO Config

The C6455 processor supports the serial RapidI/O (SRIO) peripheral from Texas Instruments for high-speed packet-switched chip-to-chip and board-to-board communications. This block provides the parameters you use to configure the SRIO peripheral on your hardware to communicate between different processors.

The dialog box parameters that you set provide values to initialize the registers on the processor relevant to SRIO processing.

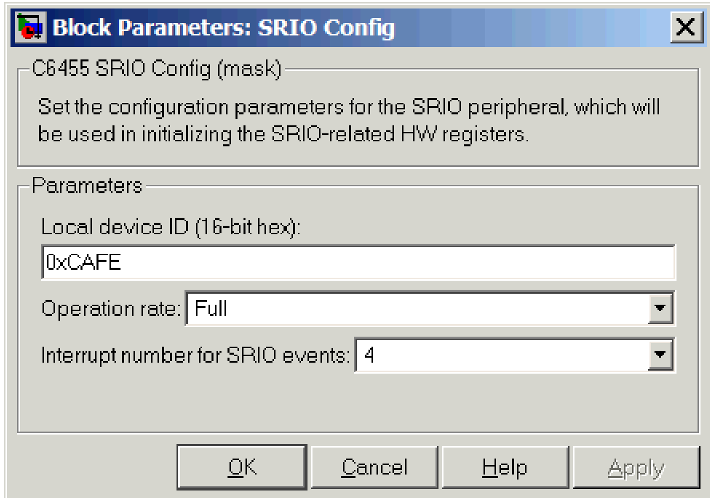
Because SRIO handles communications between two platforms, it requires two models or sets of code—one running on the local device and one running on the remote device. Both models must include the SRIO Config block to configure their SRIO communications capability, and the blocks must have the correct device IDs to refer to one another.

SRIO blocks implement both direct I/O and doorbell interrupt forms of SRIO communications. Direct I/O provides data transfer directly between two processors. With direct I/O you have burst-write and burst-read access with the remote device. The block configures the SRIO peripheral as a 4x SRIO, meaning that all four links of SRIO are bundled together for the fastest link. Direct I/O uses the Load/Store Unit (LSU) and Direct Memory Access (DMA) Engine to control and monitor the data transfer.

Doorbell interrupt enables the local device to initiate CPU interrupts on the remote device if burst-write access is enabled. Such interrupts signal that data is ready to transfer. Both devices, local (source) and remote (destination) include doorbell message queues. The destination device reads its queue to determine the interrupt source and to process the doorbell INFO field.

To see the SRIO blocks in use, refer to the Interprocessor Communications via Serial Rapid I/O (SRIO) demo, located in the online help system demos for Embedded Coder software.

Dialog Box



Local device ID (16-bit hex)

Enter the ID of the local device to configure the device ID field in the generated code. Use a 16-bit hexadecimal format. When you configure SRIO Transmit and SRIO Receive blocks in models, the local device ID in this field must match the remote device ID for the Transmit and Receive block in each model.

In the generated code, you see the input device ID as a constant mapped to the following program code entry.

```
#define SRIO_LARGE_DEV_ID 0xCAFE
```

Operation rate

Set the operating frequency of the SRIO serializer/deserializer (SERDES). Two variables determine the primary operating frequency of the SERDES, the reference clock frequency and PLL multiplication factor. Select `Full`, `Half`, or `Quarter` from the list.

- `Full` takes two data samples for each PLL output clock cycle.
- `Half` takes one data sample for each PLL output clock cycle.

- `Quarter` takes one data sample and a delay for two PLL output cycles

This value defaults to `Full`.

Interrupt number for SRIO events

Assigns an interrupt number to initiate for SRIO events. After you select a value from the list, you see a constant similar to the following defined in the generated code

```
#define SRIO_INTR_NUMBER 4
```

References

For more information about SRIO, refer to *TMS320TCI648x Serial RapidIO User's Guide*, Literature Number: SPRUE13. Texas Instruments Incorporated.

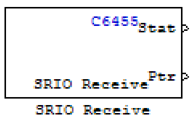
Purpose

Configure generated code to receive serial RapidI/O packets

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

Description



SRIO receive blocks add the ability to receive SRIO packets to the processor that is running the embedded code. Each receive block has two output ports—the`Stat` port that is permanent and the optional `Ptr` port, that report the status of the block and output a pointer to data.

Writing data between DSPs is more efficient than writing because SRIO write can handle up to 4kB per write request without stalling the processor while SRIO read only handles up to 256 bytes per read request. Thus, the time needed to transfer data by reading from the remote device can be much longer than that required for writing from the remote device. Use the doorbell interrupt options to signal remote devices and to coordinate the data transfer between processors.

The `Stat` port reports SRIO operating status as shown in the following table.

Value at Stat Port	Description
1	SRIO request is done (success)
0	SRIO request is pending
-1	SRIO request failed
-2	SRIO request was not sent because the SRIO request queue is full

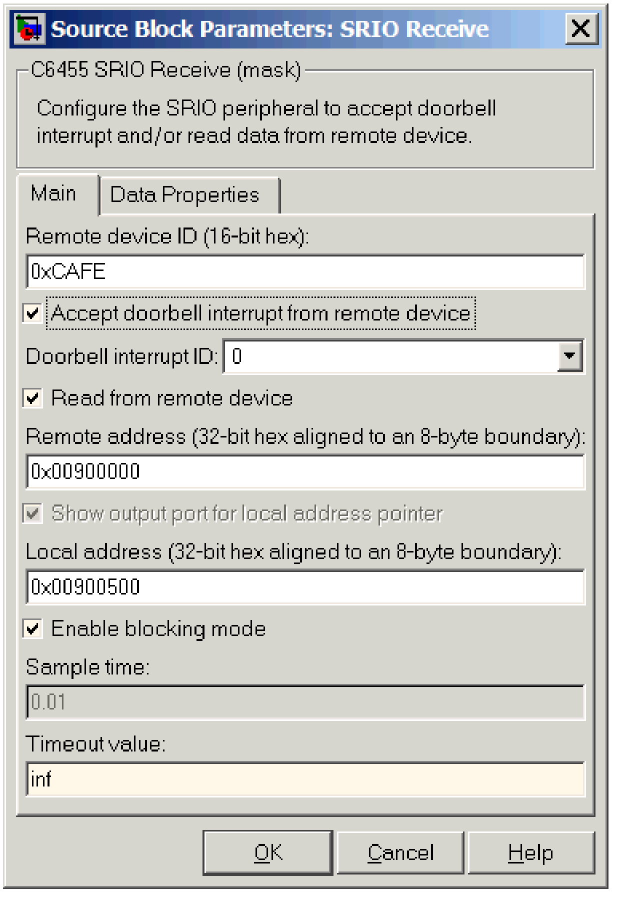
To see the SRIO blocks in use, refer to the Interprocessor Communications via Serial Rapid I/O (SRIO) demo in the online help system demos for Embedded Coder software.

Dialog Box

The block dialog box provides parameters on two panes:

- **Main** pane includes parameters that configure the data transfer operation, the doorbell interrupt ID, and various address settings for the remote device and host.
- "Data Types Pane" on page 5-360parameters configure the data type and size that the block reads.

Main Pane



Remote device ID (16-bit hex)

Enter the ID of the remote device in 16-bit hexadecimal format to configure the remote ID field in the generated code. When you configure SRIO Receive blocks for this communication link, the remote device ID in this field must match the local device ID for the SRIO Config block in the transmitting model.

Accept doorbell interrupt from remote device

Enables the doorbell interrupt operation for the block. The block always waits until it receives a doorbell interrupt before it reads from the remote device. Selecting this option enables the **Doorbell interrupt ID** parameter so you can set the interrupt ID.

Doorbell interrupt ID

Sets the interrupt ID for the doorbell to determine which SRIO Receive block should be awakened based on the incoming interrupt value. Select a value from the list. If your model contains more than one SRIO receive block, each receive block must use a different ID. IDs range from 0 to 15 with a default value of 0. SRIO Receive and SRIO Transmit blocks are paired together by this ID. Create and SRIO Transmit block with this ID to send the doorbell interrupt.

Read from remote device

Selecting this option tells the block to perform a burst read from the remote device at the address in **Remote address**. If you clear this option, you must select **Accept doorbell interrupt from remote device**.

Remote address (32-bit hex aligned to an 8-byte boundary

This address specifies where the data is being read from the remote device. The address you enter here should match the local address of the corresponding SRIO Transmit block.

This address should align to an 8-byte boundary in memory.

Show output port for local address pointer

When you select this parameter, the output port `Ptr` returns the pointer that you specify in **Local address (32-bit hex aligned to an 8 byte boundary)**. Clearing this option removes the `Ptr` port from the block.

Local address (32-bit hex aligned to an 8 byte boundary

This address specifies the destination for the data to transfer. This address should match the remote address of the corresponding

SRIO Transmit block. You will need it if the SRIO Transmit block performs burst-write operations.

Enable blocking mode

SRIO receive blocks can operate in either blocking or nonblocking modes.

- Selecting this option puts the block in blocking mode and the block waits for a doorbell interrupt to come or timeout to occur before passing program control to downstream blocks or performing any read operations.
  - Clearing **Enable blocking mode** directs the block to poll the doorbell interrupt status register to determine whether the SRIO Transmit block sent a doorbell packet.
  - Sending the packet indicates that the transmitting block completed a data transfer to this block.
- Clearing this option to put the block in nonblocking mode enables the **Sample time** option. In nonblocking mode, Simulink software uses the sample time to determine the polling period the block uses for polling the interrupt status register.

Enable blocking mode is not available when you clear **Enable doorbell**. Clearing **Accept doorbell interrupt form remote device** also disables this option because blocking mode refers to the doorbell interrupt process.

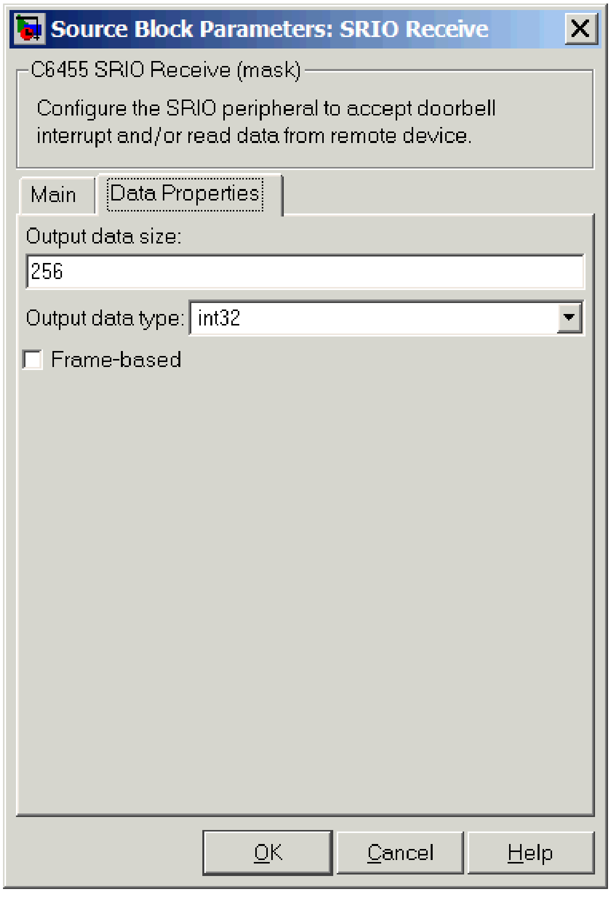
Sample time

Determines the polling period, in seconds, for the block in nonblocking mode. Enter the time period to wait between polls. To enable this option, clear **Enable blocking mode** and select **Accept doorbell interrupt from remote device**.

Timeout value

In blocking mode, this value determines how long the block waits for a doorbell interrupt before it sets the `Stat` output port to Timeout status. Enter a time in seconds (The value defaults to `inf` to block until the block receives a doorbell interrupt). The default time-out value is 1 second. Clearing either **Enable blocking mode** or **Accept doorbell interrupt from remote device** disables this option.

Data Properties Pane



Output data size

Use this to specify the amount of data in bytes to transfer. Enter either a scalar to define a vector of elements or a two-element array. For example, enter 256 to specify a vector of 256 elements. To specify a two-dimensional array of 512 elements, enter [256 2]. The block uses this value to determine the size of the `Ptr` port. If you select the **Frame-based** option, you must enter the vector, or scalar value, as an array. Thus the 256-element vector example entry becomes [256 1].

Output data type

Specify the data type used for the output. With this information, the block calculates the size of the data transfer in bytes using this value and the **Output data size** value.

Frame-based

When you select this option, the block treats the data as frame-based rather than sample-based. If you select **Frame-based**, you must enter your output data size as a two-element array. For example, to specify a vector that contains 256 elements, enter [256 1].

References

For more information about SRIO, refer to *TMS320TCI648x Serial RapidIO User's Guide*, Literature Number: SPRUE13. Texas Instruments Incorporated.

C6455 SRIO Transmit

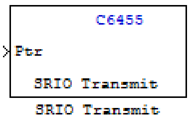
Purpose

Configure generated code to transmit serial RapidI/O packets

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6455 EVM

Description



SRIO transmit blocks add the ability to transmit SRIO packets to another processor. Each transmit block has an input `Ptr` port, and an optional `Stat` output port controlled by the **Show output port for status** option.

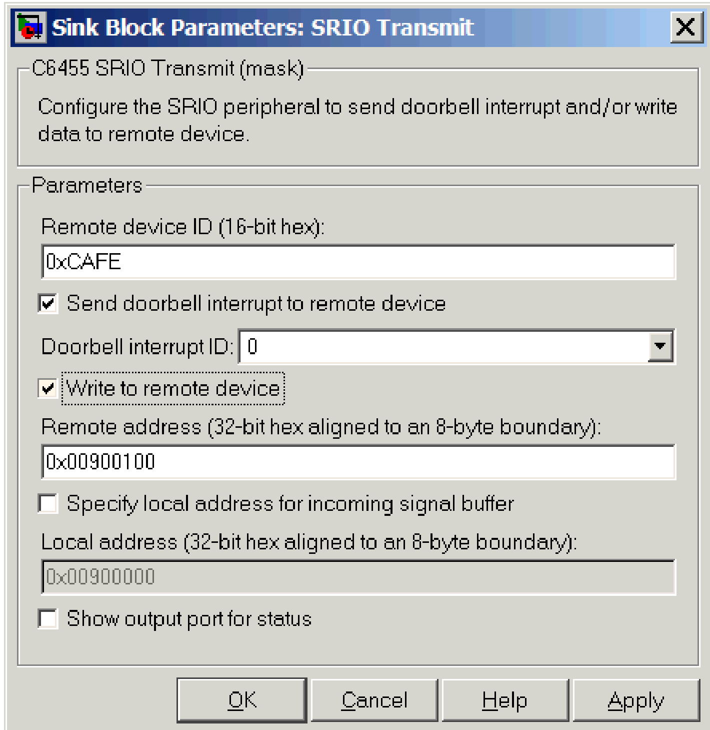
Writing data between DSPs is more efficient than reading because SRIO write can handle up to 4kB per write request without stalling the processor while SRIO read only handles up to 256 bytes per read request. Thus, the time needed to transfer data by reading from the remote device can be much longer than that required for writing from the remote device. SRIO read may require multiple requests. Use the doorbell interrupt options signal remote devices and to coordinate the data transfer between the processors.

The `Stat` port reports SRIO operating status as shown in the following table.

Value at Stat Port	Description
1	SRIO request is done (success)
0	SRIO request is pending
-1	SRIO request failed
-2	SRIO request was not sent because the SRIO request queue is full

To see the SRIO blocks in use, refer to the Interprocessor Communications via Serial Rapid I/O (SRIO) demo in the online help system demos for Embedded Coder software.

Dialog Box



Remote device ID (16-bit hex)

Enter the ID of the remote device in 16-bit hexadecimal format to configure the remote ID field in the generated code. When you configure SRIO Transmit blocks for this communication link, the remote device ID in this field must match the local device ID for the SRIO Config block on the receiving end of the transmission.

Send doorbell interrupt to remote device

Enables the doorbell interrupt operation for the bloc, which sends a doorbell interrupt after writing data to the remote device. Selecting this option enables **Doorbell interrupt ID**.

Doorbell interrupt ID

Sets the interrupt ID for the doorbell to set the doorbell INFO field of the SRIO packet. Select a value from the list. If your model contains more than one SRIO transmit block, each transmit block must use a different ID. IDs range from 0 to 15 with a default value of 0. SRIO Receive and SRIO Transmit blocks are paired together by this ID. Create an SRIO Receive block with this ID to receive the doorbell interrupt. The block uses this value to set the doorbell INFO field in an SRIO packet.

Write to remote device

Selecting this option tells the block to perform a burst write using Direct IO to the device at the address in **Remote device ID**. If you clear this option, you must select **Send doorbell interrupt to remote device**. Selecting this option enables the **Remote address (32-bit hex aligned to an 8–byte boundary** option.

Remote address (32-bit hex aligned to an 8-byte boundary

Enter the address to write the output data to at the remote device.

Clearing **Write to remote device** disables this option. It becomes and do not care field.

To ensure efficient data transfers, enter an address that aligns to an 8–byte boundary in memory.

Specify local address for incoming signal buffer

Select this option to enable you to specify the local address for the input data to this block. Select his option if you are pairing this block with an SRIO Receive block that performs burst-read operation. The SRIO Receive block needs to know the specific address to read the data from. When you select this option, you enable **Local address (32-bit hex aligned to an 8 byte boundary)** where you enter the local address.

Local address (32-bit hex aligned to an 8 byte boundary

This address specifies the location of the incoming data. For burst write operations, this value is a local address that SRIO uses to form the direct I/O packets.

To ensure efficient data transfers, enter an address that aligns to an 8–byte boundary in memory.

Show output port for status
When you select this parameter, the output port `Stat` appears on the block. `Stat` returns the status of the write transmit operation.

References

For more information about SRIO, refer to *TMS320TCI648x Serial RapidIO User's Guide*, Literature Number: SPRUE13. Texas Instruments Incorporated.

C64x Autocorrelation

Purpose

Autocorrelate input vector or frame-based matrix

Library

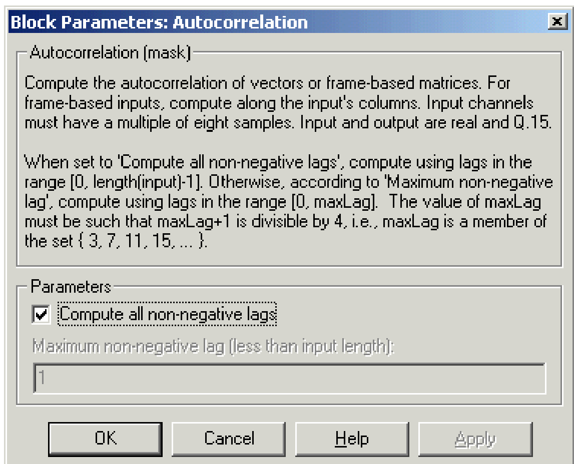
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Autocorrelation block computes the autocorrelation of an input vector or frame-based matrix. For frame-based inputs, the autocorrelation is computed along each of the input's columns. The number of samples in the input channels must be an integer multiple of eight. Input and output signals are real and Q.15.

Autocorrelation blocks support discrete sample times and little-endian code generation only.

Dialog Box



Compute all non-negative lags

When you select this parameter, the autocorrelation is performed using all nonnegative lags, where the number of lags is one less than the length of the input. The lags produced are therefore in the range [0, `length`(input)-1]. When this parameter is not selected, you specify the lags used in **Maximum non-negative lag (less than input length)**.

Maximum non-negative lag (less than input length)
Specify the maximum lag (maxLag) the block should use in performing the autocorrelation. The lags used are in the range [0, maxLag]. The maximum lag must be odd, and (maxLag+1) must be divisible by 4, such as maxLag equal to 3, 7, or 19. This parameter is enabled when you clear the **Compute all non-negative lags** parameter.

Algorithm

In simulation, the Autocorrelation block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_autocor`. During code generation, this block calls the `DSP_autocor` routine to produce optimized code.

C64x Bit Reverse

Purpose

Bit-reverse elements of each complex input signal channel

Library

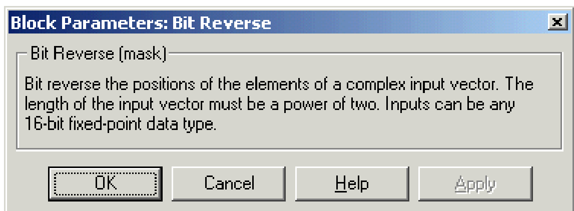
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Bit Reverse block bit-reverses the elements of each channel of a complex input signal X. The Bit Reverse block is used primarily to provide correctly-ordered inputs and outputs to or from blocks that perform FFTs. Inputs to this block must be 16-bit fixed-point data types. Input vector lengths must be a power of two. Because you use this block with FFT blocks the input vector length must be a power of two.

The Bit Reverse block supports discrete sample times and little-endian code generation only.

Dialog Box

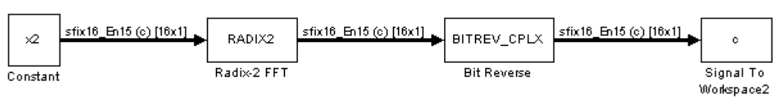


Algorithm

In simulation, the Bit Reverse block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_bitrev_cplx`. During code generation, this block calls the `DSP_bitrev_cplx` routine to produce optimized code.

Examples

The Bit Reverse block reorders the output of the C64x Radix-2 FFT in the model below to natural order.



The following code calculates the same FFT in the workspace. The output from this calculation, y2, is displayed side-by-side with the output from the model, c. The outputs match, showing that the Bit Reverse block reorders the Radix-2 FFT output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);

[y2, c]
   0.5000              0.5000
   0.4619 - 0.1913i    0.4619 - 0.1913i
   0.3536 - 0.3536i    0.3535 - 0.3535i
   0.1913 - 0.4619i    0.1913 - 0.4619i
        0 - 0.5000i         0 - 0.5000i
  -0.1913 - 0.4619i   -0.1913 - 0.4619i
  -0.3536 - 0.3536i   -0.3535 - 0.3535i
  -0.4619 - 0.1913i   -0.4619 - 0.1913i
  -0.5000             -0.5000
  -0.4619 + 0.1913i   -0.4619 + 0.1913i
  -0.3536 + 0.3536i   -0.3535 + 0.3535i
  -0.1913 + 0.4619i   -0.1913 + 0.4619i
        0 + 0.5000i         0 + 0.5000i
   0.1913 + 0.4619i    0.1913 + 0.4619i
   0.3536 + 0.3536i    0.3535 + 0.3535i
   0.4619 + 0.1913i    0.4619 + 0.1913i
```

See Also

C64x Radix-2 FFT, C64x Radix-2 IFFT

C64x Block Exponent

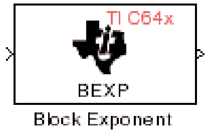
Purpose

Minimum number of extra sign bits in each input channel

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

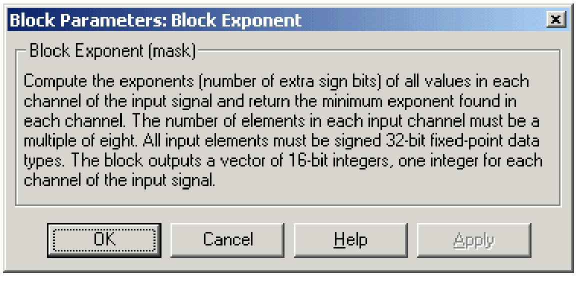


The C64x Block Exponent block first computes the number of extra sign bits of all values in each channel of an input signal, and then returns the minimum number of sign bits found in each channel. The number of elements in each input channel must be a multiple of eight. Input elements must be 32-bit signed fixed-point data types. The output is a vector of 16-bit integers — one integer for each channel of the input signal.

This block is useful for determining whether every sample in a channel is using extra sign bits. If so, you can scale your signal by the minimum number of extra sign bits to eliminate the common extra bits. This increases the representable precision and decreases the representable range of the signal.

Block Exponent blocks support both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Block Exponent block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_bexp`. During code generation, this block calls the `DSP_bexp` routine given to produce optimized code.

Purpose

Filter complex input signal using complex FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

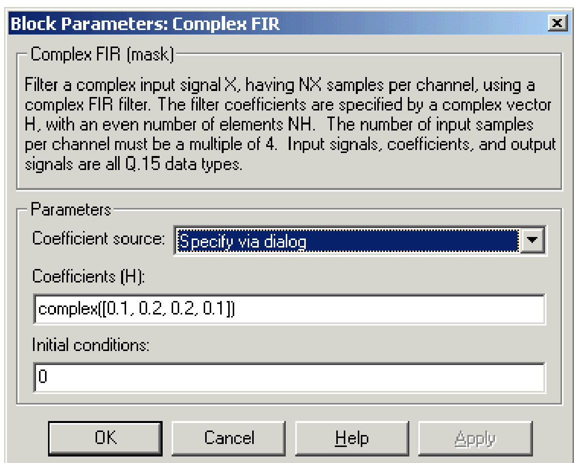
Description

The C64x Complex FIR block filters a complex input signal X using a complex FIR filter. This filter is implemented using a direct form structure. Each input channel must contain an integer multiple of four samples, with four samples as the minimum required.

The number of FIR filter coefficients, which are given as elements of the input vector H, must be even. The product of the number of elements of X and the number of elements of H must be at least four. Inputs, coefficients, and outputs are all Q.15 data types. For each channel, the number of input elements must be a multiple of four.

The Complex FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients (H)** parameter in the dialog box
- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X. Choosing this option adds an input port to the block.

Coefficients (H)

Designate the filter coefficients in vector format. There must be an even number of coefficients. This parameter is visible only when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

Lets you provide initial conditions for the filter. If your initial conditions for the channels are

- All the same, enter a scalar that applies to all channels.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. These conditions then apply to all channels. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions for every individual channel. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

  You may enter real-valued initial conditions. Zero-valued imaginary parts will be assumed.

Algorithm

In simulation, the Complex FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_cplx`. During code generation, this block calls the `DSP_fir_cplx` routine to produce optimized code.

See Also

C64x General Real FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

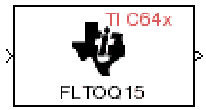
Purpose

Convert floating-point signal to Q.15 fixed-point

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

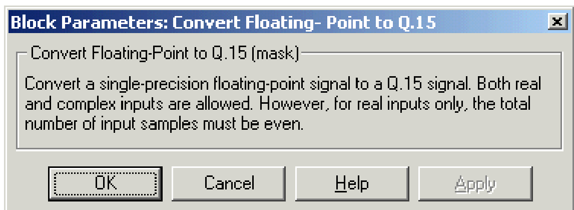


Convert Floating-Point to Q.15

The C64x Convert Floating-Point to Q.15 block converts a single-precision floating-point input signal to a Q.15 output signal. Input can be real or complex. For real inputs, the number of input samples must be even.

The Convert Floating-Point to Q.15 block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Convert Floating-Point to Q.15 block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fltoq15`. During code generation, this block calls the `DSP_fltoq15` routine to produce optimized code.

See Also

C64x Convert Q.15 to Floating Point

C64x Convert Q.15 to Floating-Point

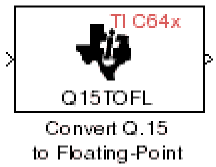
Purpose

Convert Q.15 fixed-point signal to single-precision floating-point

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

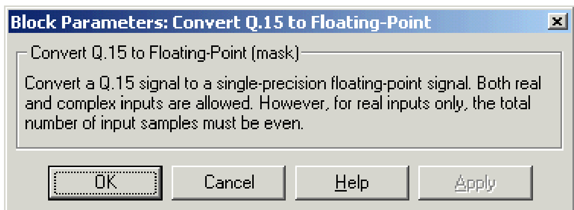
Description



The C64x Convert Q.15 to Floating-Point block converts a Q.15 input signal to a single-precision floating-point output signal. Input can be real or complex. For real inputs, the number of input samples must be even.

The Convert Q.15 to Floating-Point block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Convert Q.15 to Floating-Point block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_q15tofl`. During code generation, this block calls the `DSP_q15tofl` routine to produce optimized code.

See Also

C64x Convert Floating-Point to Q.15

Purpose

Decimation-in-frequency forward FFT of complex input vector

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

FFT

The C64x FFT block computes the decimation-in-frequency forward FFT, with scaling between stages, of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 8 to 16,384, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in natural order. Inputs and outputs are all signed 16-bit fixed-point data types.

The `fft16x16r` routine used by this block employs butterfly stages to perform the FFT. The number of butterfly stages used, S, depends on the input length L = 2^k. If k is even, then S = k/2. If k is odd, then S = (k+1)/2.

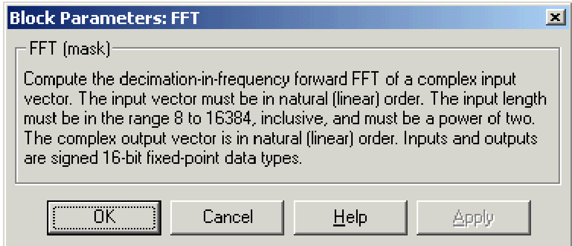
If k is even, then L is a power of two as well as a power of four, and this block performs all S stages with radix-4 butterflies to compute the output. If k is odd, then L is a power of two but not a power of four. In that case this block performs the first (S-1) stages with radix-4 butterflies, followed by a final stage using radix-2 butterflies.

To minimize noise, the FFT block also implements a divide-by-two scaling on the output of each stage except for the last. So that the gain of the block matches that of the theoretical FFT, the FFT block offsets the location of the binary point of the output data type by (S-1) bits to the right relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type minus (S-1).

$OutputFractionalBits = InputFractionalBits - (S - 1)$

The FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



Algorithm

In simulation, the FFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fft16x16r`. During code generation, this block calls the `DSP_fft16x16r` routine to produce optimized code.

See Also

C64x Radix-2 FFT, C64x Radix-2 IFFT

C64x General Real FIR

Purpose

Filter real input signal using real FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

General Real FIR

The C64x General Real FIR block filters a frame-based real input signal X using a real FIR filter. This filter is implemented using a direct form structure. Signal X must contain at least four samples per channel and the number of samples must be an integer multiple of four.

If the input it is a sample-based signal, the model throws the following error:

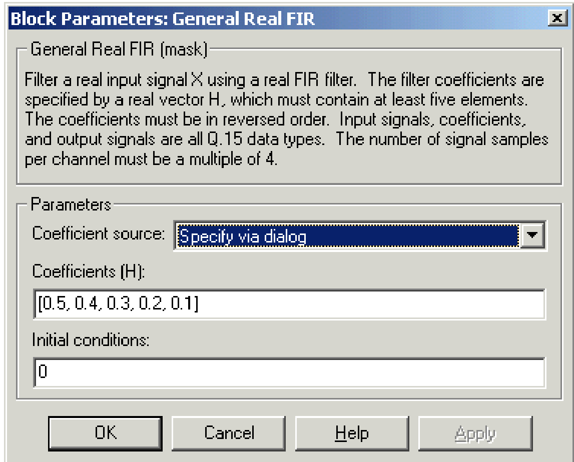
```
%%%BEGIN ERROR%%%
Error reported by S-function 'stic6x_fir_real' in 'model/General Real FIR1':
Number of output samples must be divisible by 4.
%%%END ERROR%%%
```

To resolve this error, convert the signal to a frame-based signal.

The filter coefficients are specified by a real vector H, which must contain at least five elements. The coefficients must be in reversed order. All inputs, coefficients, and outputs are Q.15 signals.

The General Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients (H)** parameter in the dialog box
- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel.

The length of this vector must be one less than the number of coefficients.

- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

The initial conditions must be real.

Algorithm

In simulation, the General Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_gen`. During code generation, this block calls the `DSP_fir_gen` routine to produce optimized code.

See Also

C64x Complex FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

C64x LMS Adaptive FIR

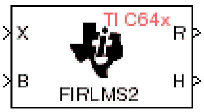
Purpose

LMS adaptive FIR filtering

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description



LMS Adaptive FIR

The C64x LMS Adaptive FIR block performs least-mean-square (LMS) adaptive filtering. This filter is implemented using a direct form structure.

Note To implement a complete LMS algorithm, use this block in combination with the 5 other blocks shown in the "Examples" on page 5-479 section.

Note This block performs fixed-point computations using fixdt(1,16,15) and fixdt(1,32,30) data types. Because of this limitation, you may not be able to address numeric overflow and underflow problems with this block. As a result, this block is useful in a limited set of applications.

The following constraints apply to the inputs and outputs of this block:

- The scalar input, X must be a Q.15 data type.
- The scalar input B must be a Q.15 data type.
- The scalar output R is a Q1.30 data type.
- The output $\bar{H}$ has length equal to the number of filter taps and is a Q.15 data type. The number of filter taps must be a positive integer that is a multiple of four.

This block performs LMS adaptive filtering according to the equations

$$e(n+1) = d(n+1) - [\bar{H}(n) \cdot \bar{X}(n+1)]$$

and

$$\bar{H}(n+1) = \bar{H}(n) + [\mu e(n+1) \cdot \bar{X}(n+1)],$$

where

- n designates the time step.
- $\bar{X}$ is a vector composed of the current and last nH–1 scalar inputs.
- d is the desired signal. The output R converges to d as the filter converges.
- $\bar{H}$ is a vector composed of the current set of filter taps.
- e is the error, or $d - [\bar{H}(n) \cdot \bar{X}(n+1)]$.
- μ is the step size.

For this block, the input B and the output R are defined by

$$B = \mu e(n+1)$$

and

$$R = \bar{H}(n) \cdot \bar{X}(n+1),$$

which combined with the first two equations, result in the following equations that this block follows:

$$e(n+1) = d(n+1) - R$$

$$\bar{H}(n+1) = \bar{H}(n) + [B \cdot \bar{X}(n+1)].$$

d and B must be produced externally to the LMS Adaptive FIR block. See “Examples” on page 5-479 below for a sample model where this is done.

The LMS Adaptive FIR block supports discrete sample times and supports little-endian code generation only.

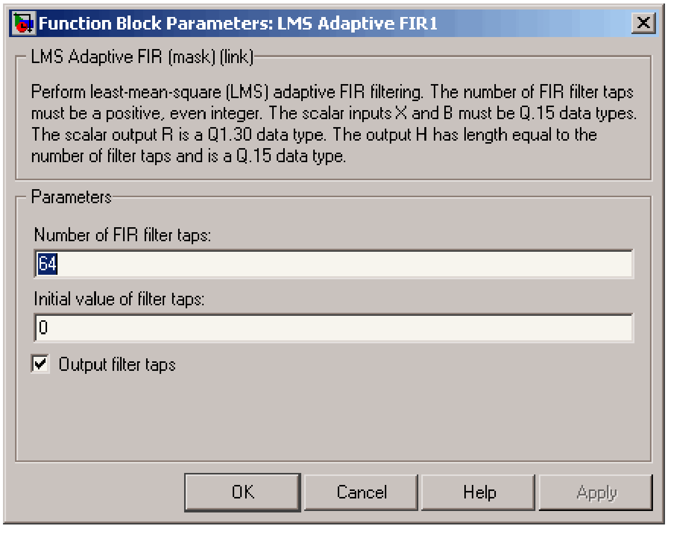
The rounding mode used is *floor*, and the saturation mode is *wrap*. All intermediate products have **s32Q30** data type. The update equation is as follows:

$$H_i = H_i + \texttt{S16Q15}(\texttt{S32Q30}(B) \times \texttt{S32Q30}(X_i))$$
$$R = \sum_N (X_i \times H_i),$$

where N is the number of filter taps.

Note This block does not implement a leaky LMS algorithm, so comparison to the leakage factor of the LMS block of the DSP System Toolbox software is not appropriate.

Dialog Box



Number of FIR filter taps
Designate the number of filter taps. The number of taps must be a positive integer that is also a multiple of four.

Initial value of filter taps
Enter the initial value of the filter taps.

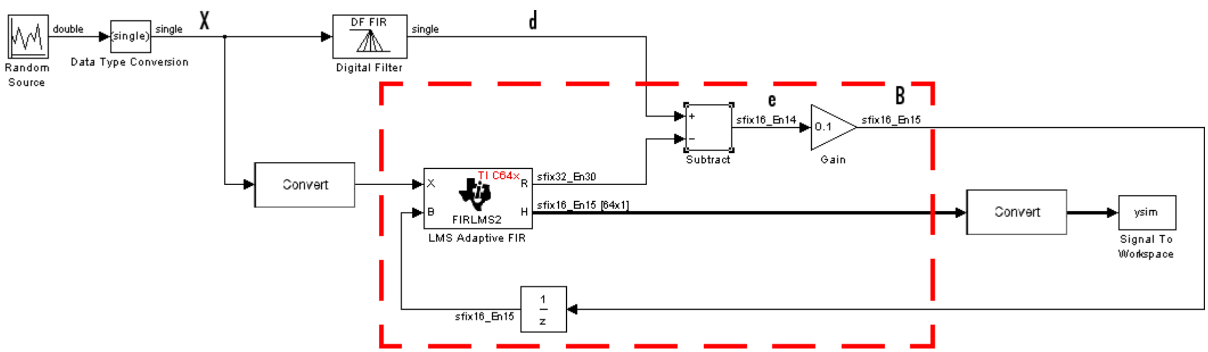
Output filter coefficients H?
If you select this option, the filter taps are produced as output H. If you do not select this option, H is suppressed.

Algorithm

In simulation, the LMS Adaptive FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_firlms2`. During code generation, this block calls the `DSP_firlms2` routine to produce optimized code.

Examples

The following model uses the LMS Adaptive FIR block.



The portion of the model enclosed by the dashed line produces the signal B and feeds it back into the LMS Adaptive FIR block. The inputs to this region are $\bar{X}$ and the desired signal d, and the output of this region is the vector of filter taps $\bar{H}$. Thus this region of the model acts as a canonical LMS adaptive filter. For example, compare this region to the `adaptfilt.lms` function in DSP System Toolbox software.

`adaptfilt.lms` performs canonical LMS adaptive filtering and has the same inputs and output as the outlined section of this model.

To use the LMS Adaptive FIR block you must create the input B in some way similar to the one shown here. You must also provide the signals $\bar{X}$ and d. This model simulates the desired signal d by feeding $\bar{X}$ into a digital filter block. You can simulate your desired signal in a similar way, or you may bring d in from the workspace with a From Workspace or codec block.

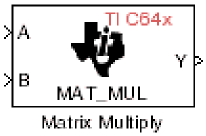
Purpose

Matrix multiply two input signals

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description



The C64x Matrix Multiply block multiplies two input matrices A and B. Inputs and outputs are real, 16-bit, signed fixed-point data types. This block wraps overflows when they occur.

The product of the two 16-bit inputs results in a 32-bit accumulator value. The Matrix Multiply block, however, only outputs 16 bits. You can choose to output the highest or second-highest 16 bits of the accumulator value.

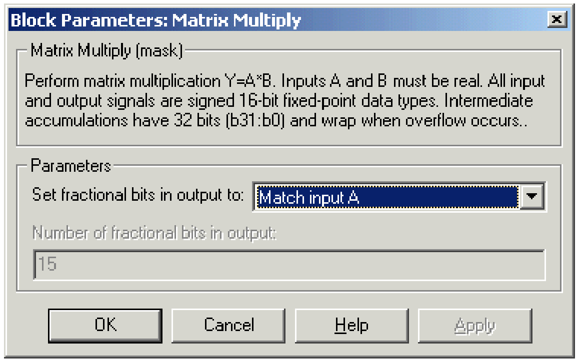
Alternatively, you can choose to output 16 bits according to how many fractional bits you want in the output. The number of fractional bits in the accumulator value is the sum of the fractional bits of the two inputs.

	Input A	Input B	Accumulator Value
Total Bits	16	16	32
Fractional Bits	R	S	$R + S$

Therefore R+S is the location of the binary point in the accumulator value. You can select 16 bits in relation to this fixed position of the accumulator binary point to give the desired number of fractional bits in the output (see "Examples" on page 5-483 below). You can either require the output to have the same number of fractional bits as one of the two inputs, or you can specify the number of output fractional bits in the **Number of fractional bits in output** parameter.

The Matrix Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Choose which 16 bits to output from the list:

- `Match input A` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input A (or R in the discussion above).
- `Match input B` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input B (or S in the discussion above).
- `Match high bits of acc. (b31:b16)` — Output the highest 16 bits of the accumulator value.
- `Match high bits of prod. (b30:b15)` — Output the second-highest 16 bits of the accumulator value.
- `User-defined` — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is enabled only when you select `User-defined` for **Set fractional bits in output to**.

Algorithm

In simulation, the Matrix Multiply block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_mat_mul`. During code generation, this block calls the `DSP_mat_mul` routine to produce optimized code.

Examples

Example 1

Suppose A and B are both Q.15. The data type of the resulting accumulator value is therefore the 32-bit data type Q1.30 ($R + S = 30$). In the accumulator, bits 31:30 are the sign and integer bits, and bits 29:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
`Match input A`	Q.15	b30:b15
`Match input B`	Q.15	b30:b15
`Match high bits of acc.`	Q1.14	b31:b16
`Match high bits of prod.`	Q.15	b30:b15

Example 2

Suppose A is Q12.3 and B is Q10.5. The data type of the resulting accumulator value is therefore Q23.8 ($R + S = 8$). In the accumulator, bits 31:8 are the sign and integer bits, and bits 7:0 are the fractional bits. The following table shows the resulting data type and accumulator bits used for the output signal for different settings of the **Set fractional bits in output to** parameter.

Set fractional bits in output to	Data Type	Accumulator Bits
`Match input A`	Q12.3	b20:b5
`Match input B`	Q10.5	b18:b3
`Match high bits of acc.`	Q23.-8	b31:b16
`Match high bits of prod.`	Q22.-7	b30:b15

See Also

C64x Vector Multiply

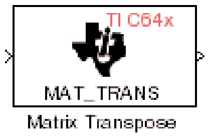
Purpose

Matrix transpose input signal

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

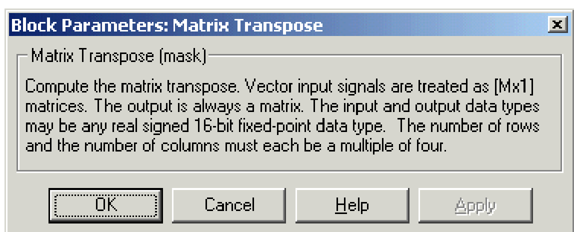


The C64x Matrix Transpose block transposes an input matrix or vector. A 1-D input is treated as a column vector and transposed to a row vector. Input and output signals are any real, 16-bit, signed fixed-point data type. Both the number of rows and the number of columns must be multiples of four.

The Matrix Transpose block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Note If you use Target Function Library (TFL) technology with this block, the TI compiler generates processor and compiler-specific instructions that improve the performance of the generated code. For more information, consult"Introduction to Target Function Libraries".

Dialog Box



Algorithm

In simulation, the Matrix Transpose block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_mat_trans`. During code generation, this block calls the `DSP_mat_trans` routine to produce optimized code.

C64x Radix-2 FFT

Purpose

Radix-2 decimation-in-frequency forward FFT of complex input vector

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

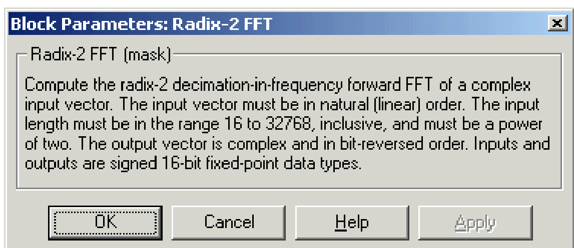
Radix-2 FFT

The C64x Radix-2 FFT block computes the radix-2 decimation-in-frequency forward FFT of each channel of a complex input signal. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types, and the output data type matches the input data type.

You can use the C64x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.

The Radix-2 FFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box

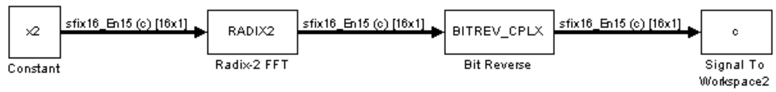


Algorithm

In simulation, the Radix-2 FFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

Examples

The output of the Radix-2 FFT block is bit-reversed. This example shows you how to use the C64x Bit Reverse block to reorder the output of the Radix-2 FFT block to natural order.



The following code calculates the same FFT as the above model in the workspace. The output from this calculation, y2, is then displayed side-by-side with the output from the model, c. The outputs match, showing that the Bit Reverse block does reorder the Radix-2 FFT block output to natural order:

```
k = 4;
n = 2^k;
xr = zeros(n, 1);
xr(2) = 0.5;
xi = zeros(n, 1);
x2 = complex(xr, xi);
y2 = fft(x2);

[y2, c]
   0.5000              0.5000
   0.4619 - 0.1913i    0.4619 - 0.1913i
   0.3536 - 0.3536i    0.3535 - 0.3535i
   0.1913 - 0.4619i    0.1913 - 0.4619i
        0 - 0.5000i         0 - 0.5000i
  -0.1913 - 0.4619i   -0.1913 - 0.4619i
  -0.3536 - 0.3536i   -0.3535 - 0.3535i
  -0.4619 - 0.1913i   -0.4619 - 0.1913i
  -0.5000             -0.5000
  -0.4619 + 0.1913i   -0.4619 + 0.1913i
  -0.3536 + 0.3536i   -0.3535 + 0.3535i
  -0.1913 + 0.4619i   -0.1913 + 0.4619i
        0 + 0.5000i         0 + 0.5000i
   0.1913 + 0.4619i    0.1913 + 0.4619i
   0.3536 + 0.3536i    0.3535 + 0.3535i
   0.4619 + 0.1913i    0.4619 + 0.1913i
```

See Also

C64x Bit Reverse, C64x FFT, C64x Radix-2 IFFT

C64x Radix-2 IFFT

Purpose

Radix-2 inverse FFT of complex input vector

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

Radix-2 IFFT

The C64x Radix-2 IFFT block computes the radix-2 inverse FFT of each channel of a complex input signal. This block uses a decimation-in-frequency forward FFT algorithm with butterfly weights modified to compute an inverse FFT. The input length of each channel must be both a power of two and in the range 16 to 32,768, inclusive. The input must also be in natural (linear) order. The output of this block is a complex signal in bit-reversed order. Inputs and outputs are signed 16-bit fixed-point data types.

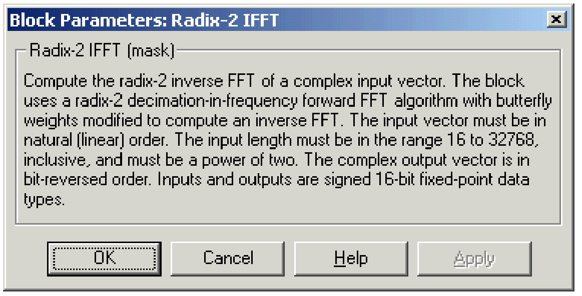
The `radix2` routine used by this block employs a radix-2 FFT of length L=2^k. So that the gain of the block matches that of the theoretical IFFT, the Radix-2 IFFT block offsets the location of the binary point of the output data type by k bits to the left relative to the location of the binary point of the input data type. That is, the number of fractional bits of the output data type equals the number of fractional bits of the input data type plus k.

$OutputFractionalBits = InputFractionalBits + (k)$

You can use the C64x Bit Reverse block to reorder the output of the Radix-2 IFFT block to natural order.

The Radix-2 IFFT block supports both continuous and discrete sample times. This block supports little-endian code generation.

Dialog Box



Algorithm

In simulation, the Radix-2 IFFT block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_radix2`. During code generation, this block calls the `DSP_radix2` routine to produce optimized code.

See Also

C64x Bit Reverse, C64x FFT, C64x Radix-2 FFT

C64x Radix-4 Real FIR

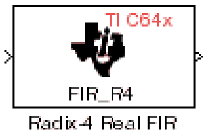
Purpose

Filter real input signal using real FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

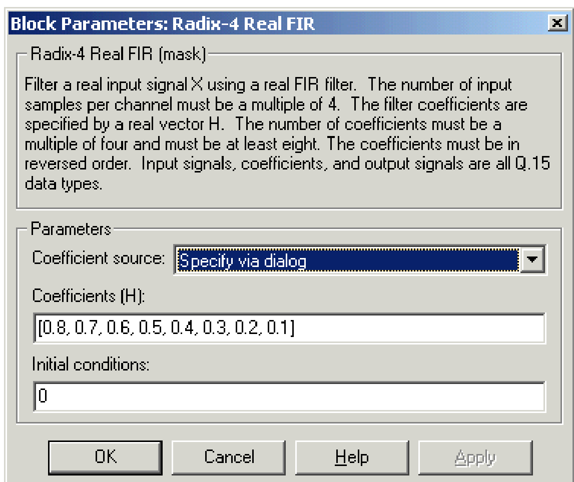


The C64x Radix-4 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be a multiple of four. The filter coefficients are specified by a real vector, H. The number of filter coefficients must be a multiple of four and must be at least eight. The coefficients must also be in reversed order {b(n), b(n-1),...,(b(0)}. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-4 Real FIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients** parameter in the dialog box

- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)

Designate the filter coefficients in vector format. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient source** parameter. Enter the *n* coefficients in reversed order — *b(n), b(n-1),...,(b(0).* This parameter is tunable in simulation.

Initial conditions

If the initial conditions are

- All the same, enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-4 Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_r4`. During code generation, this block calls the `DSP_fir_r4` routine to produce optimized code.

See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-8 Real FIR, C64x Symmetric Real FIR

C64x Radix-8 Real FIR

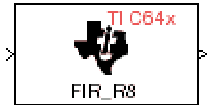
Purpose

Filter real input signal using real FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description



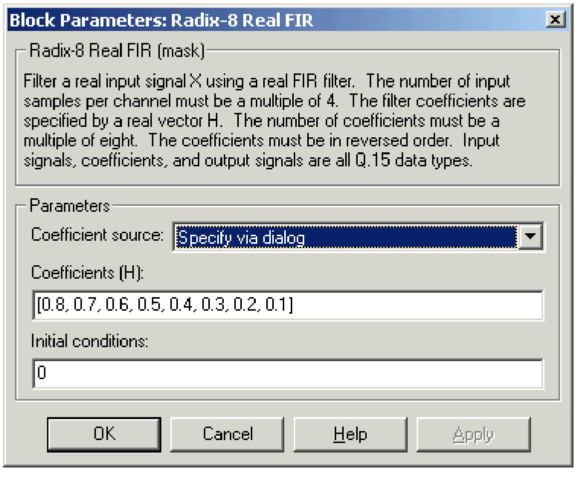
Radix-8 Real FIR

The C64x Radix-8 Real FIR block filters a real input signal X using a real FIR filter. This filter is implemented using a direct form structure.

The number of input samples per channel must be a multiple of four. The filter coefficients are specified by a real vector, H. The number of coefficients must be an integer multiple of eight. The coefficients must be in reversed order — {b(n), b(n-1),...,(b(0)}. All inputs, coefficients, and outputs are Q.15 signals.

The Radix-8 Real FIR block supports discrete sample times and little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients** parameter in the dialog box

- `Input port` — Accept the coefficients from port H. This port must have the same rate as the input data port X

Coefficients (H)
Designate the filter coefficients in vector format, entering them in reversed order — b(n), b(n-1),...,(b(0). This parameter is visible when `Specify via dialog` is selected for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions
If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Initial conditions must be real.

Algorithm

In simulation, the Radix-8 Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_r8`. During code generation, this block calls the `DSP_fir_r8` routine to produce optimized code.

See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-4 Real FIR, C64x Symmetric Real FIR

C64x Real Forward Lattice All-Pole IIR

Purpose

Filter real input signal using lattice IIR filter

Library

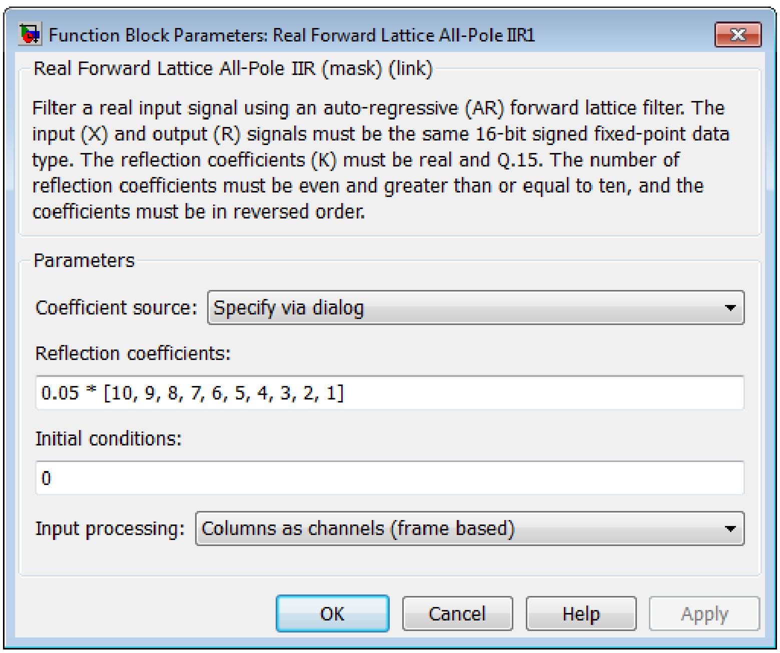
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Real Forward Lattice All-Pole IIR block filters a real input signal using an autoregressive forward lattice filter. The input and output signals must be the same 16-bit signed fixed-point data type. The reflection coefficients must be real and Q.15. The number of reflection coefficients must be greater than or equal to ten; they must be even; and they must be in reversed order — k(n), k(n-1),..., k(0). Using an even number of reflection coefficients maximizes the speed of your generated code.

The Real Forward Lattice All-Pole IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Reflection coefficients** parameter in the dialog box
- `Input port` — Accept the coefficients from port K

Reflection coefficients

Designate the reflection coefficients of the filter in vector format. The number of coefficients must be greater than or equal to ten and be even. Enter the coefficients in reverse order from k(n) to k(0). Using an even number of reflection coefficients maximizes the speed of your generated code. This parameter is visible when you select `Specify via dialog` for the **Coefficient source** parameter. This parameter is tunable in simulation.

Initial conditions

If your block initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length (number of elements) of this vector must be the same as the number of reflection coefficients in your filter.
- Different across channels, enter a matrix containing all initial conditions. The number of rows (initial conditions for one channel) of this matrix must be the same as the number of reflection coefficients, and the number of columns of this matrix must be equal to the number of channels.

Input Processing

Process input signal as frames or samples

- `Columns as channels (frame based)` — Process the input signal as frames. Each frame contains a group of sequential data samples. To perform frame-based processing, you must have a DSP System Toolbox license.

- `Elements as channels (sample based)` — Process the input signal as individual data samples.
- `Inherited (this choice will be removed   see release notes)` — Use the frame status attribute of the input signal to determine whether to process the input as frames or samples.

  When you load an existing model in R2011a, the software sets this parameter to `Inherited (this choice will be removed - see release notes)`. Selecting this option allows you to continue working with your model until you upgrade. Upgrade your model using the `slupdate` function as soon as possible.

  Note For more information about this option, see "Changes to Frame-Based Processing"

Algorithm

In simulation, the Real Forward Lattice All-Pole IIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_iirlat`. During code generation, this block calls the `DSP_iirlat` routine to produce optimized code.

See Also

C64x Real IIR

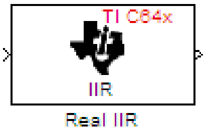
Purpose

Filter real input signal using IIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

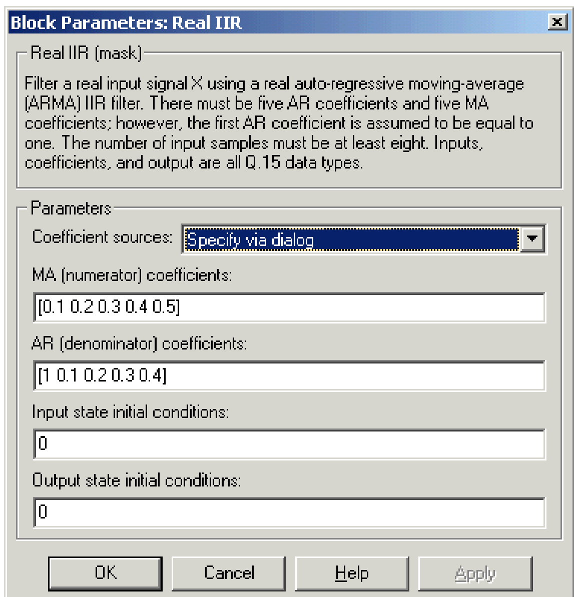


The C64x Real IIR block filters a real input signal X using a real autoregressive moving-average (ARMA) IIR Filter. This filter is implemented using a direct form I structure. You must use at least eight input samples.

There must be five AR coefficients and five MA coefficients. The first AR coefficient is always assumed to be one. Inputs, coefficients, and output are Q.15 data types.

The Real IIR block supports discrete sample times and supports little-endian code generation only.

Dialog Box



Coefficient sources

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **MA (numerator) coefficients** and **AR (denominator) coefficients** parameters in the dialog box
- `Input ports` — Accept the coefficients from ports MA and AR

MA (numerator) coefficients

Designate the moving-average coefficients of the filter in vector format. There must be five MA coefficients. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

AR (denominator) coefficients

Designate the autoregressive coefficients of the filter in vector format. There must be five AR coefficients, however the first AR coefficient is assumed to be equal to one. This parameter is only visible when `Specify via dialog` is selected for the **Coefficient sources** parameter. This parameter is tunable in simulation.

Input state initial conditions

If the input state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the input state initial conditions for one channel. The length of this vector must be four.
- Different across channels, enter a matrix containing all input state initial conditions. This matrix must have four rows.

Output state initial conditions

If the output state initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the output state initial conditions for one channel. The length of this vector must be four.

- Different across channels, enter a matrix containing all output state initial conditions. This matrix must have four rows.

Algorithm

In simulation, the Real IIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_iir`. During code generation, this block calls the `DSP_iir` routine to produce optimized code.

See Also

C64x Real Forward Lattice All-Pole IIR

C64x Reciprocal

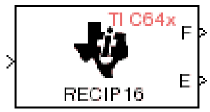
Purpose

Fraction and exponent of reciprocal of real input signal

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

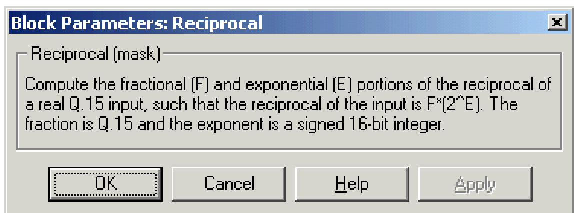


Reciprocal

The C64x Reciprocal block computes the fractional (F) and exponential (E) portions of the reciprocal of a real Q.15 input, such that the reciprocal of the input is $F*(2^E)$. The fraction is Q.15 and the exponent is a 16-bit signed integer.

The Reciprocal block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Reciprocal block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_recip16`. During code generation, this block calls the `DSP_recip16` routine to produce optimized code.

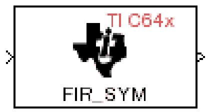
Purpose

Filter real input signal using FIR filter

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description



Symmetric Real FIR

The C64x Symmetric Real FIR block filters a real input signal using a symmetric real FIR filter. This filter is implemented using a direct form structure.

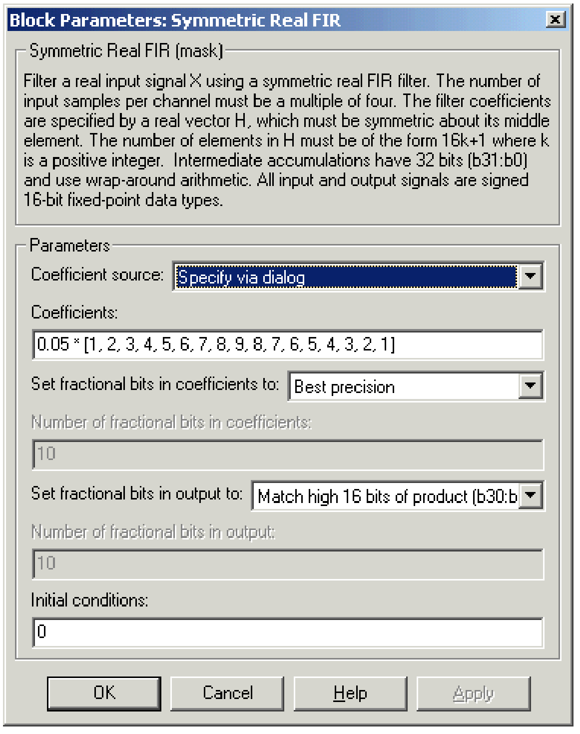
The number of input samples per channel must be even. The filter coefficients are specified by a real vector H, which must be symmetric about its middle element. Thus you must use an odd number of coefficients. The number of coefficients must be of the form 16k + 1, where k is a positive integer. This block wraps overflows that occur. The input, coefficients, and output are 16-bit signed fixed-point data types.

Intermediate multiplies and accumulates performed by this filter result in 32-bit accumulator values. However, the Symmetric Real FIR block only outputs 16 bits. You can choose to output 16 bits of the accumulator value in one of the following ways.

`Match input x`	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the input
`Match coefficients h`	Output 16 bits of the accumulator value such that the output has the same number of fractional bits as the coefficients
`Match high 16 bits of acc.`	Output bits 31 - 16 of the accumulator value
`Match high 16 bits of prod.`	Output bits 30 - 15 of the accumulator value
`User-defined`	Output 16 bits of the accumulator value such that the output has the number of fractional bits specified in the **Number of fractional bits in output** parameter

The Symmetric Real FIR block supports discrete sample times and only little-endian code generation.

Dialog Box



Coefficient source

Specify the source of the filter coefficients:

- `Specify via dialog` — Enter the coefficients in the **Coefficients** parameter in the dialog box
- `Input port` — Accept the coefficients from port H

Coefficients

Enter the coefficients in vector format. Coefficients must be symmetric about the middle element of the vector, so the number of coefficients must be odd. This parameter is visible when `Specify via dialog` is specified for the **Coefficient source** parameter. This parameter is tunable in simulation.

Set fractional bits in coefficients to

Specify the number of fractional bits in the filter coefficients:

- `Match input X` — Sets the coefficients to have the same number of fractional bits as the input
- `Best precision` — Sets the number of fractional bits of the coefficients such that the coefficients are represented to the best precision possible
- `User-defined` — Sets the number of fractional bits in the coefficients with the **Number of fractional bits in coefficients** parameter

This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter.

Number of fractional bits in coefficients

Specify the number of bits to the right of the binary point in the filter coefficients. This parameter is visible only when `Specify via dialog` is specified for the **Coefficient source** parameter, and is only enabled if `User-defined` is specified for the **Set fractional bits in coefficients to** parameter.

Set fractional bits in output to

Only 16 bits of the 32 accumulator bits are output from the block. Select which 16 bits to output:

- `Match input X` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in input X
- `Match coefficients H` — Output the 16 bits of the accumulator value that cause the number of fractional bits in the output to match the number of fractional bits in coefficients H

- `Match high bits of acc. (b31:b16)` — Output the highest 16 bits of the accumulator value
- `Match high bits of prod. (b30:b15)` — Output the second-highest 16 bits of the accumulator value
- `User-defined` — Output the 16 bits of the accumulator value that cause the number of fractional bits of the output to match the value specified in the **Number of fractional bits in output** parameter

See Matrix Multiply "Examples" on page 5-483 for demonstrations of these selections.

Number of fractional bits in output

Specify the number of bits to the right of the binary point in the output. This parameter is only enabled if `User-defined` is selected for the **Set fractional bits in output to** parameter.

Initial conditions

If the initial conditions are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be one less than the number of coefficients.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be one less than the number of coefficients, and the number of columns of this matrix must be equal to the number of channels.

Algorithm

In simulation, the Symmetric Real FIR block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_fir_sym`. During code generation, this block calls the `DSP_fir_sym` routine to produce optimized code.

See Also

C64x Complex FIR, C64x General Real FIR, C64x Radix-4 Real FIR, C64x Radix-8 Real FIR

C64x Vector Dot Product

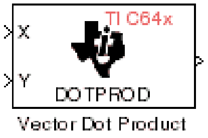
Purpose

Vector dot product of real input signals

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

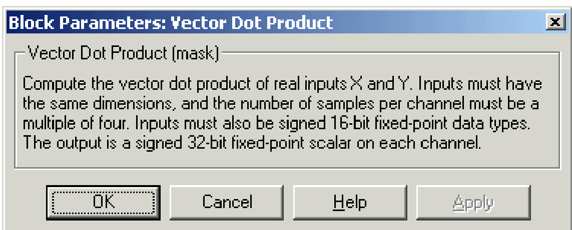
Description



The C64x Vector Dot Product block computes the vector dot product of two real input vectors, X and Y. The input vectors must have the same dimensions and must be signed 16-bit fixed-point data types. The number of samples per channel of the inputs must be a multiple of four. The output is a signed 32-bit fixed-point scalar on each channel, and the number of fractional bits of the output is equal to the sum of the number of fractional bits of the inputs.

The Vector Dot Product block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Dot Product block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_dotprod`. During code generation, this block calls the `DSP_dotprod` routine to produce optimized code.

Purpose

Zero-based index of maximum value element in each input signal channel

Library

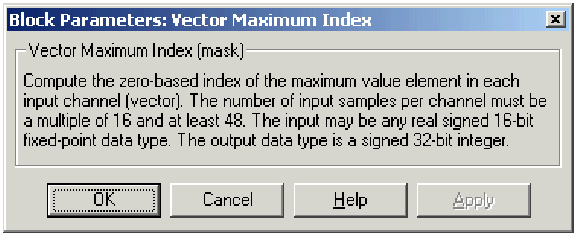
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Vector Maximum Index block computes the zero-based index of the maximum value element in each channel (vector) of the input signal. The input may be any real, 16-bit, signed fixed-point data type. The number of samples per input channel must be an integer multiple of 16 and at least 48. The output data type is 32-bit signed integer.

The Vector Maximum Index block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Maximum Index block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_maxidx`. During code generation, this block calls the `DSP_maxidx` routine to produce optimized code.

C64x Vector Maximum Value

Purpose

Maximum value for each input signal channel

Library

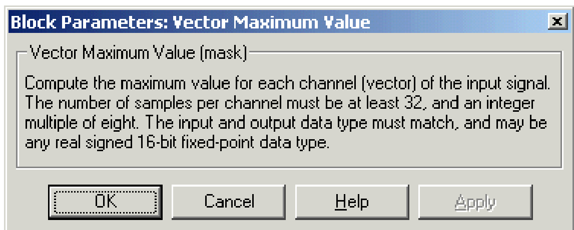
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Vector Maximum Value block returns the maximum value in each channel (vector) of the input signal. The input can be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of 8 and must be at least 32. The output data type matches the input data type.

The Vector Maximum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Maximum Value block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_maxval`. During code generation, this block calls the `DSP_maxval` routine to produce optimized code.

See Also

C64x Vector Minimum Value

Purpose

Minimum value for each input signal channel

Library

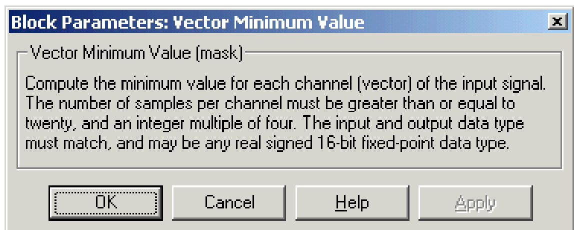
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Vector Minimum Value block returns the minimum value in each channel of the input signal. The input may be any real, 16-bit, signed fixed-point data type. The number of samples on each input channel must be an integer multiple of 4 and must be at least 20. The output data type matches the input data type.

The Vector Minimum Value block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Minimum Value block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_minval`. During code generation, this block calls the `DSP_minval` routine to produce optimized code.

See Also

C64x Vector Maximum Value

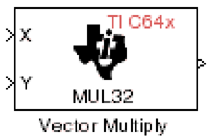
Purpose

Element-wise multiplication on inputs

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

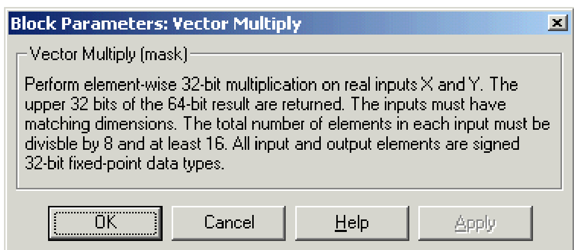
Description



The C64x Vector Multiply block performs element-wise 32-bit multiplication of two inputs X and Y. The total number of elements in each input must be a multiple or 8 and at least 16, and the inputs must have matching dimensions. The upper 32 bits of the 64-bit accumulator result are returned. All input and output elements are 32-bit signed fixed-point data types.

The Vector Multiply block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Multiply block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_mul32`. During code generation, this block calls the `DSP_mul32` routine to produce optimized code.

See Also

C64x Matrix Multiply

C64x Vector Negate

Purpose

Negate each input signal element

Library

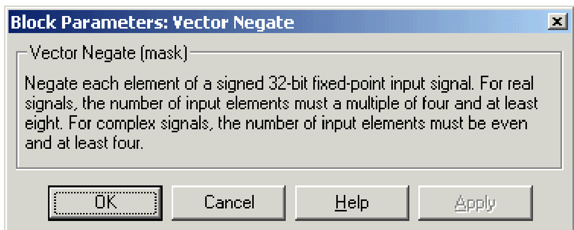
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

Description

The C64x Vector Negate block negates each element of a 32-bit signed fixed-point input signal. For real signals, the number of input elements must be a multiple of four, and at least eight. For complex signals, the number of input elements must be at least two. The output is the same data type as the input.

The Vector Negate block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Negate block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_neg32`. During code generation, this block calls the `DSP_neg32` routine to produce optimized code.

C64x Vector Sum of Squares

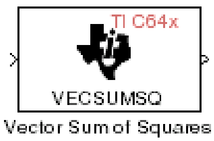
Purpose

Sum of squares over each real input channel

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

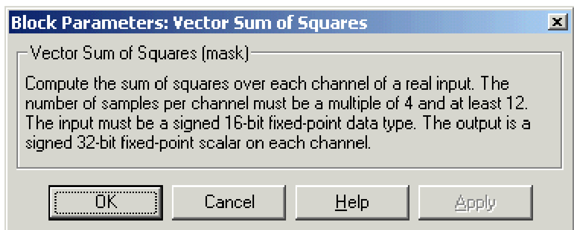
Description



The C64x Vector Sum of Squares block computes the sum of squares over each channel of a real input. The number of samples per input channel must be divisible by 4; equal to or greater than 8; and the input must be a 16-bit signed fixed-point data type. The output is a 32-bit signed fixed-point scalar on each channel. The number of fractional bits of the output is twice the number of fractional bits of the input.

The Vector Sum of Squares block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Algorithm

In simulation, the Vector Sum of Squares block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_vecsumsq`. During code generation, this block calls the `DSP_vecsumsq` routine to produce optimized code.

Purpose

Weighted sum of input vectors

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Optimization/ C64x DSP Library

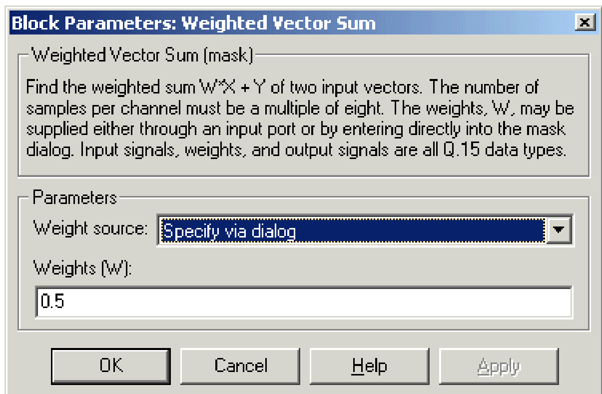
Description

Weighted Vector Sum

The C64x Weighted Vector Sum block computes the weighted sum of two inputs, X and Y, according to (W*X)+Y. Inputs may be vectors or frame-based matrices. The number of samples per channel must be a multiple of eight. Inputs, weights, and output are Q.15 data types, and weights must be in the range -1 < W < 1.

The Weighted Vector Sum block supports both continuous and discrete sample times. This block supports little-endian code generation only.

Dialog Box



Weight source

Specify the source of the weights:

- `Specify via dialog` — Enter the weights in the **Weights (W)** parameter in the dialog box
- `Input port` — Accept the weights from port W

Weights (W)

This parameter is visible only when `Specify via dialog` is specified for the **Weight source** parameter. This parameter is tunable in simulation. When the weights are

- All the same, you need only enter a scalar.
- Different within channels but the same across channels, enter a vector containing the initial conditions for one channel. The length of this vector must be a multiple of four.
- Different across channels, enter a matrix containing all initial conditions. The number of rows of this matrix must be a multiple of four, and the number of columns of this matrix must be equal to the number of channels.

  Weights must be in the range -1 < W < 1.

Algorithm

In simulation, the Weighted Vector Sum block is equivalent to the TMS320C64x DSP Library assembly code function `DSP_w_vec`. During code generation, this block calls the `DSP_w_vec` routine to produce optimized code.

Purpose

Digitized signal output from codec to processor

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6713 DSK

Description

Use the C6713 DSK ADC (analog-to-digital converter) block to capture and digitize analog signals from external sources, such as signal generators, frequency generators or audio devices. Placing an C6713 DSK ADC block in your Simulink block diagram lets you use the audio coder-decoder module (codec) on the C6713 DSK to convert an analog input signal to a digital signal for the digital signal processor.

Due to a hardware limitation, there can be only one C6713 DSK ADC block per model. Using two blocks will generate an error message.

Most of the configuration options in the block affect the codec. However, the **Output data type**, **Samples per frame** and **Scaling** options are related to the model you are using in Simulink software, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each option listed with the C6713 DSK hardware affected.

Option	Affected Hardware
ADC source	Codec
Mic	Codec
Output data type	TMS320C6713 digital signal processor
Samples per frame	Direct memory access functions
Scaling	TMS320C6713 digital signal processor
Source gain (dB)	Codec

You can select one of three input sources from the **ADC source** list:

- `Line In` — the codec accepts input from the line in connector (LINE IN) on the board's mounting bracket.
- `Mic` — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels on input and output.

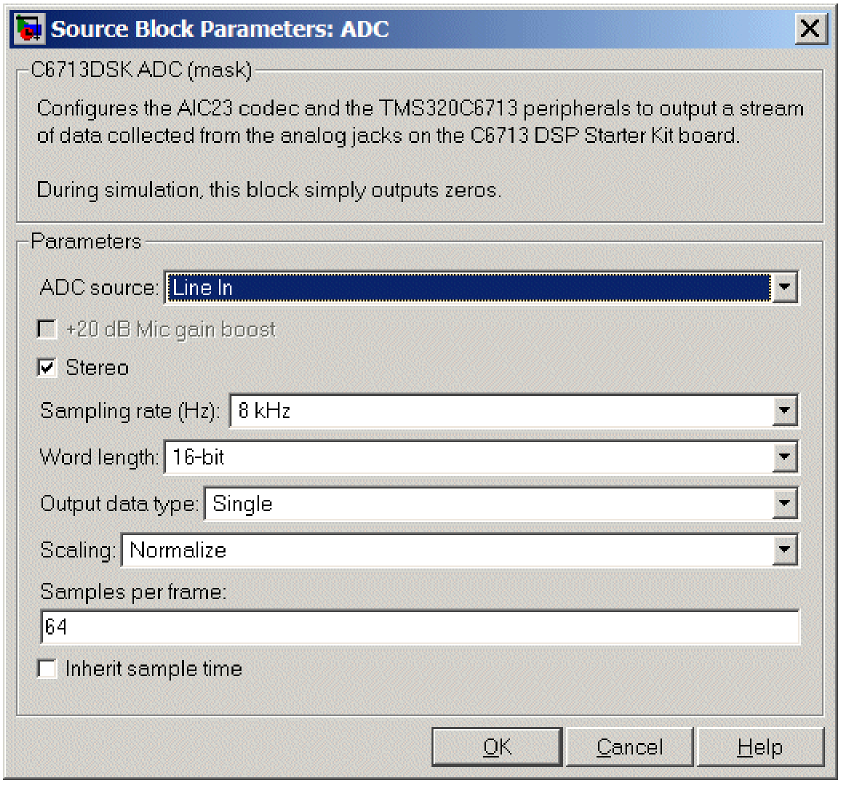
The block uses frame-based processing of inputs, buffering the input data into frames at the specified samples per frame rate. In Simulink software, the block puts monaural data into an N-element column vector. Stereo data input forms an N-by-2 matrix with N data values and two stereo channels (left and right).

When the samples per frame setting is more than one, each frame of data is either the N-element vector (monaural input) or N-by-2 matrix (stereo input). For monaural input, the elements in each frame form the column vector of input audio data. In the stereo format, the frame is the matrix of audio data represented by the matrix rows and columns — the rows are the audio data samples and the columns are the left and right audio channels.

When you select `Mic` for **ADC source**, you can select the **+20 dB Mic gain boost** check box to add 20 dB to the microphone input signal before the codec digitizes the signal.

Source gain (dB) lets you add gain to the input signal before the A/D conversion. Select the appropriate gain from the list.

Dialog Box



ADC source

The input source to the codec. `Line In` is the default setting. Selecting `Mic` enables the **+20 dB Mic gain boost** option.

+20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is `Mic`. Gain is applied before analog-to-digital conversion.

Stereo

Indicates whether the input audio data is in monaural or stereo format. Select the check box to enable stereo input. Clear the check box when you input monaural data. By default, stereo operation is enabled.

Sampling Rate
Set the sampling rate of the analog-to-digital converter. Increasing the frequency increases the accuracy of the sampling data over time.

Word length
Sets the resolution with which the ADC samples the analog input. Increasing the word length increases the accuracy of the data in each sample. If your model also contains a DAC block, set its word length match that of the ADC block.

Output data type
Selects the word length and shape of the data from the codec. By default, `double` is selected. Options are `Double`, `Single`, and `Integer`.

Scaling
Selects whether the codec data is unmodified, or normalized to the output range to ±1.0, based on the codec data format. Select either `Normalize` or `Integer Value`. `Normalize` is the default setting.

Samples per frame
Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal the block 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. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 8kHz samples per second, and you select 64 samples per frame, the frame rate is 125 frames every second. The throughput remains the same at 64 samples per second.

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.

See Also

C6713 DSK DAC

C6713 DSK DAC

Purpose

Configure codec to convert digital input to analog output

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6713 DSK

Description

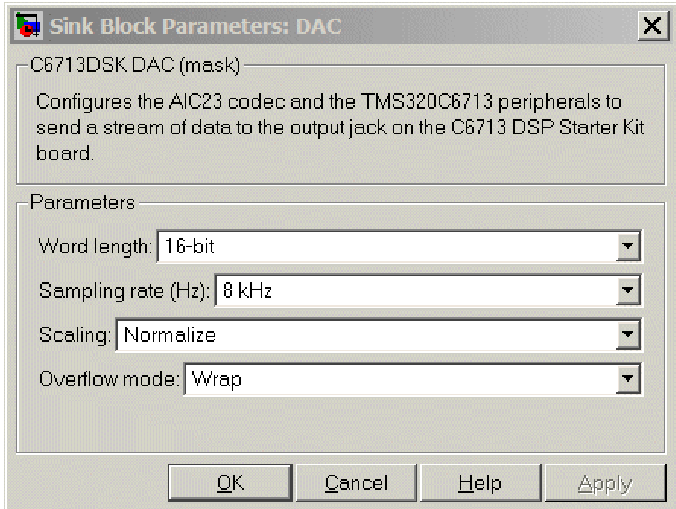
Adding the C6713 DSK DAC (digital-to-analog converter) block to your Simulink model lets you connect an analog signal to the analog output jack on the C6713 DSK. When you add the C6713 DSK DAC block, the digital signal received by the codec is converted to an analog signal and sent to the output jack.

The input on the C6713DSK DAC block takes [Nx1] and [Nx2] signals. The AIC23 audio codec on the C6713DSK board always outputs stereo samples, even though it accepts both mono [Nx1] and stereo [Nx2] signals. If the input is a mono signal with dimension [Nx1], the block outputs the same signal on both the left and right channels. If the input is a stereo signal with dimension [Nx2], each of the N samples are output separately through the left and right channels.

Only the **Word length** option in the block affects the codec. The other options relate to the model you are using in Simulink software and the signal processor on the board. Refer to the following table for information.

Option	Affected Hardware
Overflow mode	TMS320C6713 Digital Signal Processor
Scaling	TMS320C6713 Digital Signal Processor
Word length	Codec

Dialog Box



Word length

Sets the DAC to interpret the input data word length. Without this setting, the DAC cannot convert the digital data to analog correctly. The value defaults to 16 bits, with options of 20, 24, and 32 bits. Select the word length to match the ADC setting.

Scaling

Selects whether the input to the codec represents unmodified data, or data that has been normalized to the range ±1.0. Matching the setting for the C6713 DSK ADC block is appropriate here.

Overflow mode

Determines how the codec responds to data that is outside the range specified by the **Scaling** parameter. You can choose `Wrap` or `Saturate` options to apply to the result of an overflow in an operation. `Saturation` is the less efficient operating mode if efficiency is important to your development.

See Also

C6713 DSK ADC

C6713 DSK DIP Switch

Purpose

Simulate or read DIP switches

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6713 DSK

Description

Added to your model, this block behaves differently in simulation than in code generation and targeting.

In Simulation — the options **Switch 0**, **Switch 1**, **Switch 2,** and **Switch 3** generate output to simulate the settings of the user-defined dual inline pin (DIP) switches on your C6713 DSK. Each option turns the associated DIP switch on when you select it. The switches are independent of one another.

By defining the switches to represent actions on your target, DIP switches let you modify the operation of your process by reconfiguring the switch settings.

Use the **Data type** to specify whether the DIP switch options output an integer or a logical string of bits to represent the status of the switches. The table that follows presents all the option setting combinations with the result of your **Data type** selection.

Option Settings to Simulate the User DIP Switches on the C6713 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Cleared	Cleared	Cleared	0000	0
Selected	Cleared	Cleared	Cleared	0001	1
Cleared	Selected	Cleared	Cleared	0010	2
Selected	Selected	Cleared	Cleared	0011	3
Cleared	Cleared	Selected	Cleared	0100	4
Selected	Cleared	Selected	Cleared	0101	5

Option Settings to Simulate the User DIP Switches on the C6713 DSK (Continued)

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Cleared	Selected	Selected	Cleared	0110	6
Selected	Selected	Selected	Cleared	0111	7
Cleared	Cleared	Cleared	Selected	1000	8
Selected	Cleared	Cleared	Selected	1001	9
Cleared	Selected	Cleared	Selected	1010	10
Selected	Selected	Cleared	Selected	1011	11
Cleared	Cleared	Selected	Selected	1100	12
Selected	Cleared	Selected	Selected	1101	13
Cleared	Selected	Selected	Selected	1110	14
Selected	Selected	Selected	Selected	1111	15

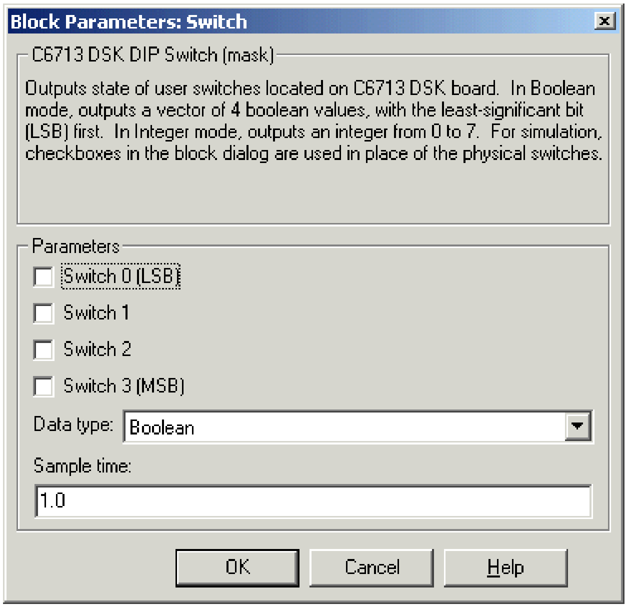
Selecting the `Integer` data type results in the switch settings generating integers in the range from 0 to 15 (`uint8`), corresponding to converting the string of individual switch settings to a decimal value. In the `Boolean` data type, the output string presents the separate switch setting for each switch, with the **Switch 0** status represented by the least significant bit (LSB) and the status of **Switch 3** represented by the most significant bit (MSB).

In Code generation and targeting — the code generated by the block reads the physical switch settings of the user switches on the board and reports them as shown above. Your process uses the result in the same way whether in simulation or in code generation. In code generation and when running your application, the block code ignores the settings for **Switch 0**, **Switch 1**, **Switch 2** and **Switch 3** in favor of reading the hardware switch settings. When the block reads the DIP switches, it reports the results as either a Boolean string or an integer value as the table below shows.

Output Values From The User DIP Switches on the C6713 DSK

Switch 0 (LSB)	Switch 1	Switch 2	Switch 3 (MSB)	Boolean Output	Integer Output
Off	Off	Off	Off	0000	0
On	Off	Off	Off	0001	1
Off	On	Off	Off	0010	2
On	On	Off	Off	0011	3
Off	Off	On	Off	0100	4
On	Off	On	Off	0101	5
Off	On	On	Off	0110	6
On	On	On	Off	0111	7
Off	Off	Off	On	1000	8
On	Off	Off	On	1001	9
Off	On	Off	On	1010	10
On	On	Off	On	1011	11
Off	Off	On	On	1100	12
On	Off	On	On	1101	13
Off	On	On	On	1110	14
On	On	On	On	1111	15

Dialog Box



Switch 0

Simulate the status of the user-defined DIP switch on the board.

Switch 1

Simulate the status of the user-defined DIP switch on the board.

Switch 2

Simulate the status of the user-defined DIP switch on the board.

Switch 3

Simulate the status of the user-defined DIP switch on the board.

Data type

Determines how the block reports the status of the user-defined DIP switches. `Boolean` is the default, indicating that the output is a vector of four logical values, either 0 or 1.

Each vector element represents the status of one DIP switch; the first switch is switch **Switch 0** and the fourth is switch **Switch 3**.

The data type `Integer` converts the logical string to an equivalent unsigned 8-bit (`uint8`) value. For example, when the logical string generated by the switches is 0101, the conversion yields 5 — the LSB is 1 and the MSB is 0.

Sample time

Specifies the time between samples of the signal. This value defaults to 1 second between samples, for a sample rate of one sample per second (1/**Sample time**).

Purpose

Control LEDs

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6713 DSK

Description

Adding the C6713 DSK LED block to your Simulink block diagram lets you trigger all four of the user light emitting diodes (LED) on the C6713 DSK. To use the block, send a nonzero real scalar to the block. The C6713 DSK LED block controls all four User LEDs located on the C6713 DSK.

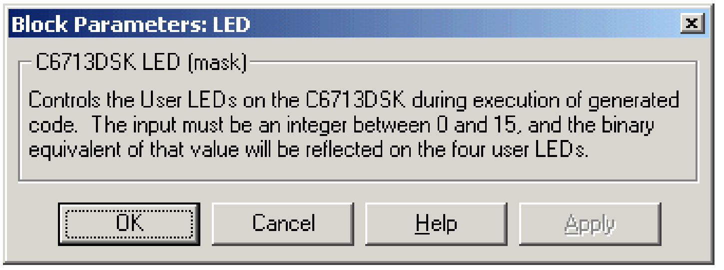
When you add this block to a model, and send a real scalar to the block input, the block sets the LED state based on the input value it receives:

- When the block receives an input value equal to 0, the specified LEDs are turned off (disabled), 0000
- When the block receives a nonzero input value, the specified LEDs are turned on (enabled), 0001 to 1111

To activate the block, send it an integer in the range 0 to 15. Vectors do not work to activate LEDs; nor do complex numbers as scalars or vectors.

All LEDs maintain their state until they receive an input value that changes the state. Enabled LEDs stay on until the block receives an input value that turns the LEDs off; disabled LEDs stays off until turned on. Resetting the C6713 DSK turns off all User LEDs. By default, the LEDs are turned off when you start an application.

Dialog Box



This dialog box does not have any user-selectable options.

Purpose

Reset to initial conditions

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6713 DSK

Description

Double-clicking this block in a Simulink model window resets the C6713 DSK that is running the executable code built from the model. When you double-click the Reset block, the block runs the software reset function provided by CCS IDE that resets the processor on your C6713 DSK. Applications running on the board stop and the signal processor returns to the initial conditions you defined.

Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library it resets your C6713 DSK. In other words, anytime you double-click a C6713 DSK Reset block you reset your C6713 DSK.

Dialog Box

This block does not have settable options and does not provide a user interface dialog box.

C6000 CPU Timer

Purpose

Select timer and configure periodic interrupt

Library

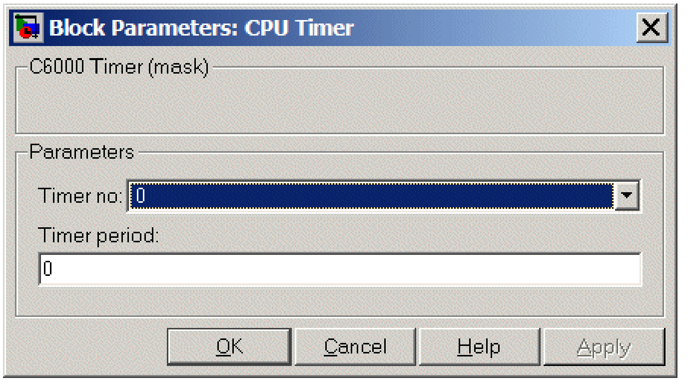
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description

Configures the CPU timer period on your board. The timer raises periodic interrupts when the timer counter reaches the timer period. While the block provides two timers, 0 and 1, some CPU's have more or fewer than two timers. For example, the DM642 provides three timers. If you set **Timer no** to **1**, verify that your CPU has two or more timers.

The C6000 CPU Timer block does not support C64x processors.

Dialog Box



Timer no.
: Select the timer to use from the list. Verify that the target offers a timer with the timer number you choose. Timer 0 is selected by default.

Timer period
: Set the timer interrupt period in terms of CPU clock cycles.

Enter the timer period in clock cycles, either as an integer, fraction, decimal, or a variable in your workspace. `0` is the default value.

For example, to generate a periodic timer interrupt every second when the CPU clock operates at 720MHz, set **Timer period** to `720e6` clock cycles.

See Also

C5000/C6000 Hardware Interrupt, Idle Task

DM642 EVM Audio ADC

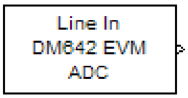
Purpose

Audio codec and peripherals

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description



Audio ADC

Use the DM642 EVM ADC (analog-to-digital converter) block to capture and digitize analog audio signals from external sources, such as signal generators, frequency generators, or audio devices. Placing a DM642 EVM ADC block in your Simulink block diagram lets you use the audio coder-decoder module (codec) on the DM642 EVM to convert an analog input signal to a digital signal for the digital signal processor.

ADC blocks output `int16` data independent of the data type you provide as input to the block.

Most of the configuration options in the block affect the codec. However, the **Samples per frame** and **Scaling** options are related to the model you are using in Simulink software, the signal processor on the board, or direct memory access (DMA) on the board. In the following table, you find each option listed with the DM642 EVM hardware affected.

Option	Affected Hardware
ADC Source	Codec
Mic	Codec
Sample rate (Hz)	Codec
Samples per frame	Direct memory access functions
Stereo	Codec

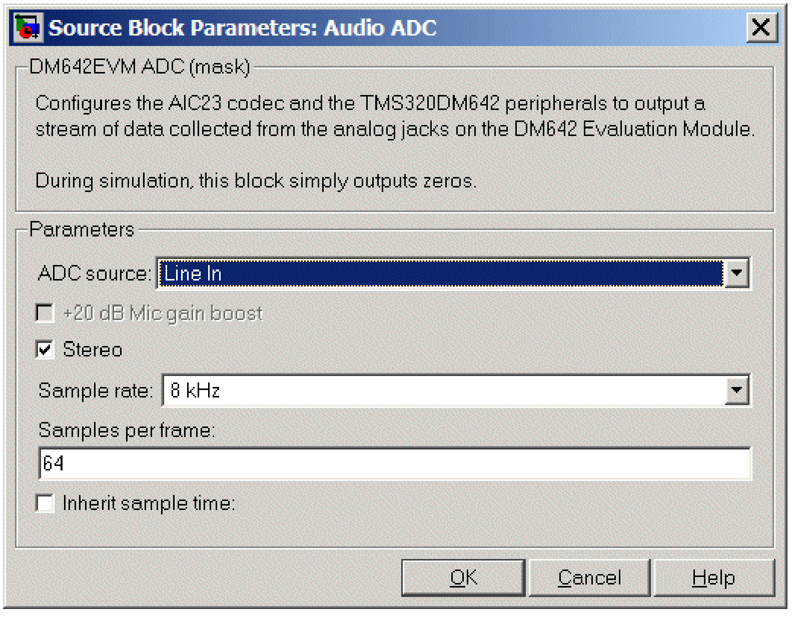
You can select one of two input sources from the **ADC source** list:

- **Line In** — the codec accepts input from the line in connector (LINE IN) on the board's mounting bracket.
- **Mic in** — the codec accepts input from the microphone connector (MIC IN) on the board mounting bracket.

Use the **Stereo** check box to indicate whether the audio input is monaural or stereo. Clear the check box to choose monaural audio input. Select the check box to enable stereo audio input. Monaural (mono) input is left channel only, but the output sends left channel content to both the left and right output channels; stereo uses the left and right channels.

You must set the sample rate for the block. From **Sample rate (Hz)**, select the sample rate for your model. **Sample rate (Hz)** specifies the number of times each second that the codec samples the input signal. Sample rates range from 8 kHz to 96 kHz, in preset rates. You must select from the list; you cannot enter a sample rate that is not on the list.

Dialog Box



ADC source

The input source to the codec. `Line In` is the default.

+20 dB Mic gain boost

Boosts the input signal by +20dB when **ADC source** is `Mic`. Gain is applied before analog-to-digital conversion.

Stereo

The number of channels input to the A/D converter. Clearing this option selects the left channel; selecting this option selects both left and right input channels. To configure the DM642 EVM board for monaural operation, clear the **Stereo** check box. When you first open the dialog box, **Stereo** is selected. This value defaults to stereo operation.

Sample rate (Hz)

Sampling rate of the A/D converter. Available sample rates are set by the codec. Default rate is `8` kHz. Options range up to `96` kHz. Select the sample rate from the list.

Samples per frame

Creates frame-based outputs from sample-based inputs. This parameter specifies the number of samples of the signal buffered internally by the block 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. Notice that the frame rate depends on the sample rate and frame size. For example, if your input is 32 samples per second, and you select 64 samples per frame, the frame rate is one frame every two seconds. The throughput remains the same at 32 samples per second.

Inherit sample time

Selects whether the block inherits the sample time from the model base rate or Simulink base rate as determined in the `Solver` options in Configuration Parameters. Selecting **Inherit sample time** directs the block to use the specified rate in model configuration. You must select this option to use the block in a function subsystem with the asynchronous scheduler.

See Also

DM642 EVM Audio DAC

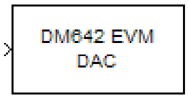
Purpose

Configure codec to convert digital audio input to analog audio output

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description



Audio DAC

Adding the DM642 EVM DAC (digital-to-analog converter) block to your Simulink model lets you output an analog signal to the LINE OUT connection on the DM642 EVM mounting bracket. When you add the DM642 EVM DAC block, the digital signal received by the codec is converted to an analog signal (digital-to-analog conversion) and sent to the output audio jack.

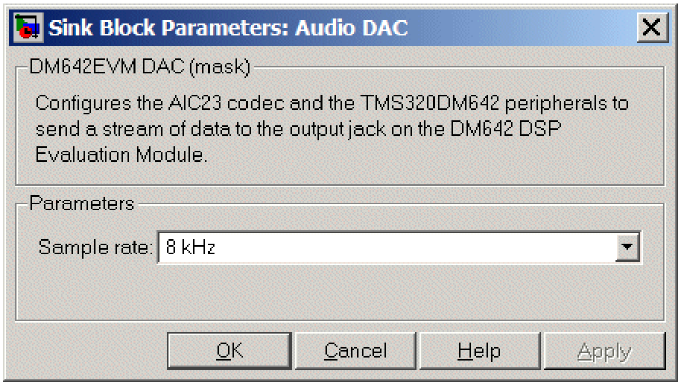
The DAC data word length is 16 bits. The block converts all input data to `int16` before it writes the data out to the DAC output buffer.

With an integer data word length of 16 bits, any data value above $2^{15}-1$ or below -2^{15} wraps back into the representable range of values between -2^{15} to $2^{15}-1$. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. Saturate arithmetic is not available. For example,

While converting the digital signal to an analog signal, the codec rounds floating point data to the nearest integer, thus rounding 0.51 up to 1.0 or 4.49 down to 4.0.

Setting the sample rate configures the codec sampling rate for the analog output data stream. The rates range from 8000 Hz, similar to plain old telephone service quality, to 48 kHz (CD quality audio) to 96 kHz.

Dialog Box



Sample rate (Hz)

Sampling rate of the D/A converter. Available output sample rates are set by the codec. Default rate is 8000 Hz (`8 kHz`) and the maximum rate is 96000 Hz (`96 kHz`). Choose the appropriate rate from the list.

See Also

DM642 EVM Audio ADC

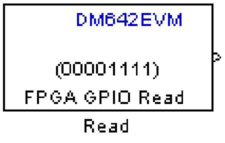
Purpose

User GPIO registers to read from selected pins

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description

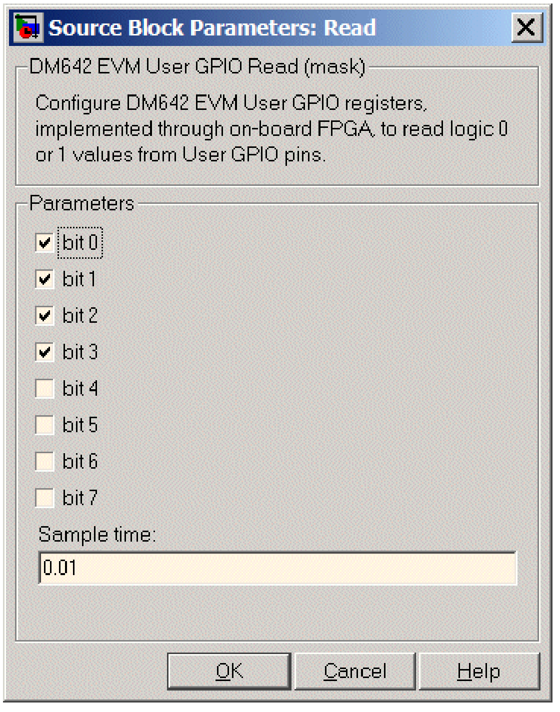


Added to your model, this block reads logical values from the GPIO registers you select in the dialog box and sends the data out to downstream blocks as an unsigned 8-bit word.

The DM642 EVM offers eight general purpose I/O registers that you can read from and write to for your needs. Each I/O pin represents either a logical 0 or 1 depending on the signal at the pin.

An important note — you cannot read and write to the same I/O registers with the FPGA GPIO Read and FPGA GPIO Write blocks. If you read register 1 with the read block you cannot write to register 1 with the write block. This applies to all eight registers.

Dialog Box



bit 0 to bit 7

Each bit represents the logical value at one GPIO register. **Bit 0** is register 0, **bit 7** is register 7. Select the bits that represent the registers to read. The read and write functions cannot share the same registers. If you select a register to read, you cannot write to that register.

Sample time

Time in seconds between consecutive inputs to the registers. Enter any real positive value or a variable name from your workspace.

See Also

DM642 EVM FPGA GPIO Write

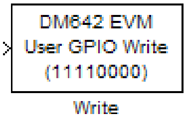
Purpose

Write to GPIO registers

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description

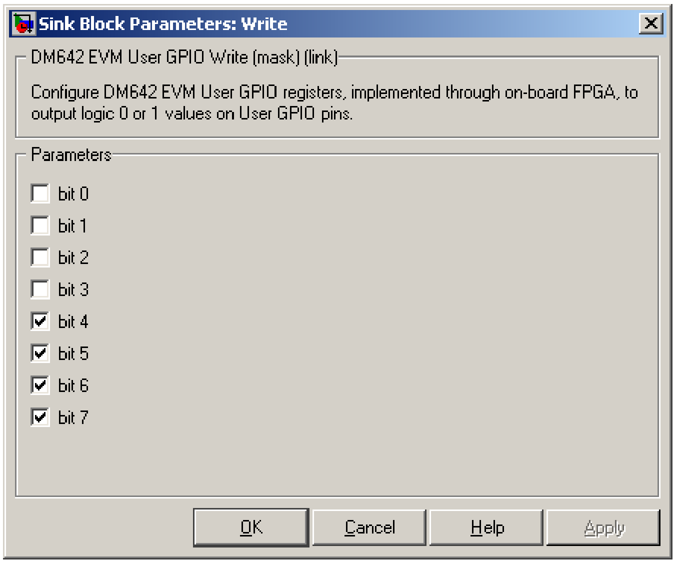


Added to your model, this block writes logical values to the GPIO registers you select in the dialog box, reading the data from an upstream block as an unsigned 8-bit word.

The DM642 EVM offers eight general purpose I/O registers that you can read from and write to for your needs. Each I/O pin represents either a logical 0 or 1 depending on the signal at the pin.

An important note — you cannot read and write to the same I/O registers with the FPGA GPIO Read and FPGA GPIO Write blocks. If you write register 1 with the write block you cannot read from register 1 with the read block. This applies to all eight registers.

Dialog Box



bit 0 to bit 7

Each bit represents the logical value at one GPIO register. **Bit 0** is register 0, **bit 7** is register 7. Select the bits that represent the registers to write. The read and write functions cannot share the same registers. When you select a register to write to, you cannot read that register.

See Also

DM642 EVM FPGA GPIO Read

Purpose

Video decoders to capture analog video

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description

Adding this block to a model enables code generated from your model to perform the following tasks:

1. Capture analog video data from the video input ports on the DM642 EVM.

2. Convert the input to a format and mode you define in the block.

3. Output the converted digital video for further downstream processing.

Adding two of these blocks to a model lets you capture two separate video data streams and prepare them for display simultaneously, such as in picture-in-picture mode.

The block captures and buffers one frame (two fields for NTSC standard) of analog input video from the input ports, converts the buffered video to the specified format, and then outputs the converted video frame as 8-bit unsigned integer data for further processing.

Input to the DM642 EVM must be analog National Television Standards Committee (NTSC) or Phase Alternating Line (PAL) video format. The block captures and processes data in frames, not fields.

To configure the format for the output video, the block offers output format options that control how the block handles color data. The block also offers a sample time option to let you set the frame rate for video output from the block.

Note This block does not provide output video for display. Use the DM642 EVM Video DAC to generate video data to output to the board video output connectors. The DM642EVM board provides both composite and S-video connectors for output. However, these are driven simultaneously, so you do not need to specify which one is to be used.

When you add this block to a Simulink model, it has no affect in your simulation — it outputs a string of zeros. Generating code from a model that includes this block produces the code needed for capturing data on your evaluation module by adding

- Video device configuration code for the chosen mode
- Code used to copy the run time buffer

To use video in a Simulink model, use one of the available video source blocks to introduce video data to your model.

Options for the block let you configure the digital video format and video mode for the data output by the block.

NTSC TV systems use interlaced scanning to create TV frames from fields. The even and odd TV lines are separated into even and odd fields that combine to make a complete TV frame image. For output, the block always provides complete frames, consisting of two fields, which are available at any instant. When the sample time you specify for the block is different from the NTSC frame rate of 30Hz, you may encounter visible anomalies in the video stream from the block.

Memory Use

This block allocates video capture buffers on the system heap, using a TI driver that allocates three frame buffers on the heap for continuous video capture. To use the block you must create a heap in external memory on the target with the label EXTERNALHEAP. If you do not create the heap, either using the default values in the DM642 Target Preferences block or setting your own values. Embedded Coder software returns an error.

Use **Create heap** and **Heap size** and set the heap size in the DM642EVM Target Preferences block to configure the heap. Select **Define label** and name the heap EXTERNALHEAP in **Heap label**.

The default settings for the Target Preferences create a heap with sufficient memory to handle the worst case memory allocation needs automatically. If you configure the heap without sufficient memory, you get a run-time error because the system cannot initialize the video driver.

Notes About Converting NTSC Video Input From YCbCr to RGB24

When you choose to convert your NTSC YCbCr-defined video input to RGB24 (8:8:8 RGB) for output from the block, the block performs an intermediate conversion step that follows a standard process for conversion (as described by Graphical Device Interface (GDI) color space conversions documentation from the International Color Consortium (ICC)).

First, the block converts the luma component (Y'), blue-difference chroma component (Cb), and red-difference chroma component (Cr) of the input signal to 5:6:5 RGB format where the red and blue channels of the source use a 5-bit representation and the green channel uses 6 bits.

Now the block converts your 5:6:5 RGB to 8:8:8 RGB using the following conventions:

1. For the red and blue 5-bit channels, it copies the three most significant bits (MSB) from the 5-bit source word and append them to the lower order end of the target word.

2. For the green 6-bit channel, it copies the two MSBs from the green source word and append them to the lower order end of the target green word.

The results is to output three RGB channels — red, green, and blue — each with 8-bit words.

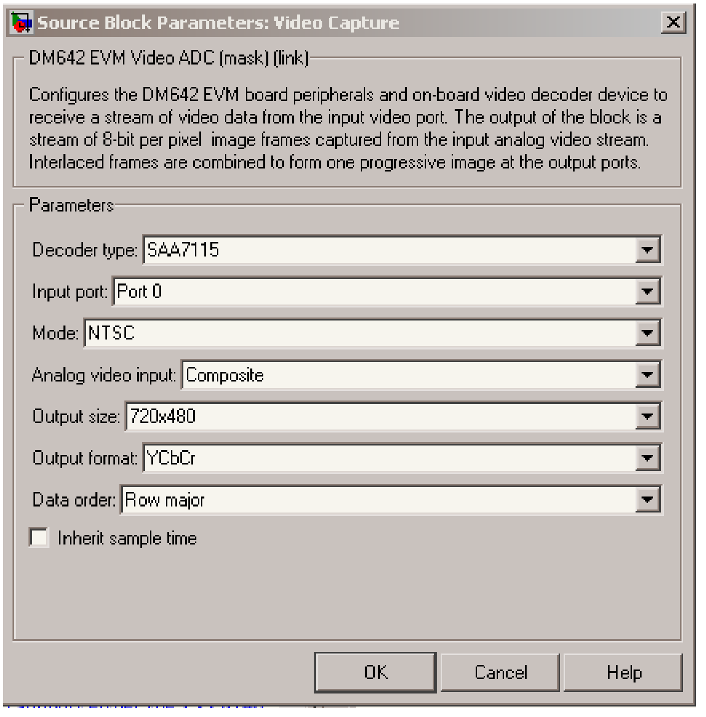
For example, to convert hexadecimal values by this algorithm, 5:5:5 RGB data of (0x19, 0x33, 0x1A) becomes (0xCE, 0xCF, 0xD6) of 8:8:8 RGB output.

To do the conversion in the binary case for 5:5:5 RGB data:

1. blue data 1 1101 converts to 11101111
2. for the green channel, conversion takes 11 0011 to 1100 1111
3. red data 1 0101 becomes 1010 1101 (same algorithm as blue data)

To maximize the speed of the RGB conversion, the Video ADC block provides color space conversion using a routine written in assembly language and optimized for the DM64x processor core. Using the optimized color space conversion code replaces the Color Space Conversion block available from the Computer Vision System Toolbox™ (VIP blockset). While you can use any compatible VIP blockset block with the DM642, this particular color space conversion operation is handled better by the conversion code included in the ADC block.

Dialog Box



Decoder type
: Configures the block options to support either the TVP5146 Decoder on the DM642 EVM or the SAA7115 Decoder, depending on the model of your board. Choose one option from the list — `TVP5146` or `SAA7115`. When you select SAA7115 for the type of decoder, the dialog box adds a new option — **Output Mode**. Generally, older DM642 EVM boards use the SAA7115 decoder. Newer boards use the default setting TVP5146 decoder.

Input port
: Directs the block to capture video from either the 0 or 1 video input port on the DM642 EVM. The block does not support port 2 for video input. Input port 0 provides both composite video (via connector J15) and S-video (connector J16) inputs.

Mode

Select the video format to capture from the list. The block supports NTSC and PAL video formats.

Analog Video Input

Select composite video or S-video. The video decoder connected to port 0 has both composite and S-video inputs. These are available via connector J15 and J16, respectively. Port 1 has two composite video connectors and no S-video availability.

Output size

Reports the size of the video images to output. **Output size** is a read-only parameter set to `720 x 576` resolution elements when you select PAL mode and the TVP5146 decoder in **Decoder type**. When you select NTSC mode with the TVP5146 decoder, **Output size** reports the read-only value `720 x 480`.

If you select the SAA7115 decoder, **Output size** lists the available video sizes to output for further processing, depending on the **Mode** setting. The following tables show the sizes to pick from depending on whether you pick NTSC or PAL for **Mode** The block scales the input video to the selected size for output.

Video Output Size Options For NTSC Mode	Description
`128 x 96`	Output NTSC video with dimensions 128 pixels by 96 pixels. Scales the output to 1/4 the resolution of QCIF video.
`176 x 144`	Output NTSC video with dimensions 176 pixels by 144 pixels. Scales the output to 1/4 the resolution of CIF video.

Video Output Size Options For NTSC Mode	Description
320 x 240	Output NTSC video with dimensions 320 pixels by 240 pixels. Scales the output to standard interchange format NTSC. Derived from CCIR 601 video (most often).
720 x 480	Output NTSC video with dimensions 720 pixels by 480 pixels. Scales the output to higher definition TV mode.

Video Output Size Options For PAL Mode	Description
128 x 96	Output video with dimensions 128 pixels by 96 pixels
176 x 144	Output video with dimensions 176 pixels by 144 pixels.
320 x 240	Output video with dimensions 320 pixels by 240 pixels
720 x 576	Output video with dimensions 720 pixels by 576 pixels

Output format

Determines how the block represents color data in the output. Choose one of the following color representations according to what your model and algorithm require.

Digital Output Format	Description
RGB24	Output uses 8 bits each of red, green, and blue colors to represent the color of each pixel in the image. RGB color space is device-dependent.
YCbCr	Output from the block includes three channels to represent the color image data per pixel: • Y — the luma component (essentially a black/white signal) • Cb — the blue-difference chroma component • Cr — the red-difference chroma component This is the digital standard color space DVDs use.
Y	Black/White video. No color/chromaticity values.

Data order

With data order, you control the way the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

- `Row major` — store video data in row major order. This is the default setting and matches most video data.
- `Column major` — store video data in column major order. The Simulink and MATLAB software use this format to store images and matrices.

DM642 EVM Video ADC blocks store the image data in row major format because most video capture devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

MATLAB and Simulink software use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video ADC blocks.

To address this problem, the Video ADC blocks include an option **Data order** to let you select either row major or the column major storage formats. By default, this block uses row major data format.

When you select `Column major`, the block performs an explicit transposition on the image data to map the data format from row major to column major order. To minimize the processor time spent on the transposition, the block uses optimized assembly routines to transpose the image data.

Inherit sample time

Selecting **Inherit sample time** sets the sample time to –1. To use this block in a function call subsystem, you must select this option. **Inherit sample time** is cleared by default and the block uses the model sample time.

Specifying sample-time inheritance for a this block, a source block, can cause Simulink software to assign an inappropriate sample time to the block. You should avoid selecting **Inherit sample time** unless you are required to do so because you placed the block in a function call subsystem. When you select **Inherit sample time**, Simulink software displays a warning message when you update or simulate the model.

See Also

DM642 EVM Video DAC

DM642 EVM Video DAC

Purpose

Video encoder to display video

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description

In the project generated from a model, this block provides the code to gather video from another block in the model, and direct the video stream to the video output port on the board.

You should input unsigned 8-bit integers to the block in the specified mode.

Adding this block to a model enables code generated from your model to perform the following tasks:

1. Capture digital video data from the application on your DM642 EVM.
2. Buffer the captured video into frames for NTSC display — two fields per frame and 30 frames per second, or SVGA display — RGB24 color with noninterlaced frames.
3. Convert to analog video.
4. Output the converted analog video to the EVM Video Out ports.

Unlike the DM642 EVM Video ADC block, this DAC block does not convert the video between formats. Nor does this block inherit any settings from the DM642 EVM Video ADC block, as some of the other C6000 DAC blocks do.

The **Mode** option specifies both the video format the block accepts and the format the block outputs to the video output ports on the EVM.

To be able to be displayed, images that you send to the block should be equal to or smaller than the target display size. If the input images are smaller than the target display size, the block pads the image by adding zeros to the image.

When you add this block to your Simulink model, it has no affect on your simulation — it outputs a string of zeros. In code generation, the block creates the device code needed to buffer, convert, and send video to the output port on the EVM.

Note The DM642EVM board provides both composite and S-video connectors for output. However, these are driven simultaneously, so you do not need to specify which one is to be used.

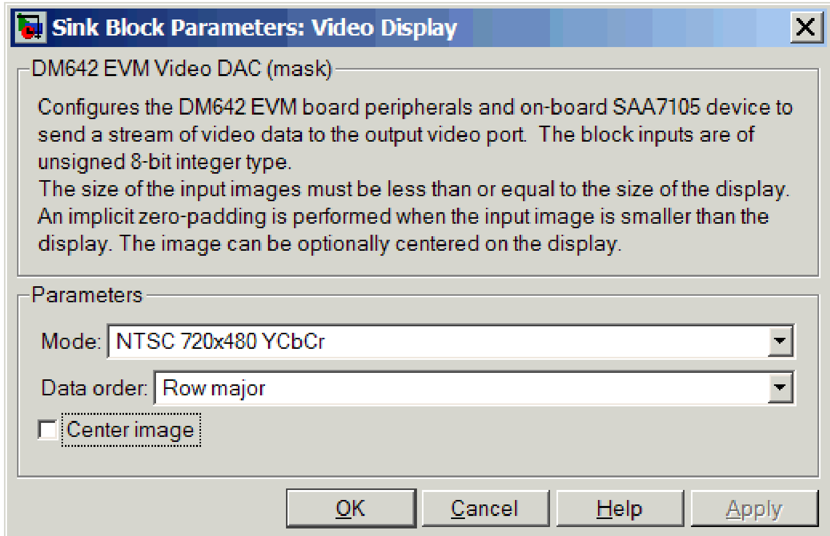
Memory Use

This block allocates video capture buffers on the system heap, using a TI driver that allocates three frame buffers on the heap for continuous video capture. To use the block you must create a heap in external memory on the target with the label EXTERNALHEAP. If you do not create the heap, either using the default values in the DM642 Target Preferences block or setting your own values. Embedded Coder software returns an error.

Use **Create heap** and **Heap size** and set the heap size in the DM642EVM Target Preferences block to configure the heap. Select **Define label** and name the heap EXTERNALHEAP in **Heap label**.

The default settings for the Target Preferences create a heap with sufficient memory to handle the worst case memory allocation needs automatically. If you configure the heap without sufficient memory, you get a run-time error because the system cannot initialize the video driver.

Dialog Box



Mode

Specifies the video format for the block. The block then sends video in this format to the video output port on the EVM. The **Mode** parameter offers the following options:

Analog Output Mode	Description
`NTSC 720x480 YCbCr`	Analog output of video data in 720-by-480 pixels format with full color.
`NTSC 640x480 Y`	Analog video output in 640-by-480 pixels format with black and white only (luminance). No color data.
`SVGA 800x600 RGB24`	Full super VGA format 800-by-600 pixels with three color channels: 8-bit red, 8-bit green, and 8-bit blue data.

Analog Output Mode	Description
`PAL 720x570 YCbCr`	Analog output of video data in 720-by-570 pixels PAL format with full color.
`PAL 720 x 570 Y`	Analog output of video data in 720-by-570 pixels PAL format with black and white only (luminance). No color data.

Data order

With data order, you control the way the video decoder stores and outputs video data fields and frames of images. Choose one of these options from the list.

- `Row major` — store video data in row major order. This is the default setting and matches most video data.
- `Column major` — store video data in column major order. Simulink and MATLAB software use this format to store images and matrices.

  DM642 EVM Video DAC blocks store the image data in row major format because most video display devices use a scanning order of left-to-right and top-to-bottom, favoring the rows.

  MATLAB and Simulink software use column major ordering to store image and matrix data. Therefore, some of the Simulink blocks may not work correctly or as expected with the DM642 EVM Video DAC blocks.

  To address this problem, the Video DAC blocks include an option **Data order** to let you select either row major or the column major storage formats. By default, these blocks use row major data format.

  When the column major data ordering option is selected, the block performs an explicit transposition on the image data to map the data format from row major to column major order.

To minimize the processor time spent on the transposition, the block uses optimized assembly routines to accomplish the image transposition.

Center Image

Directs the block to center the output image on the display. Centering the image requires some computation by the processor so there are small time and CPU cycles penalties for choosing this option. For that reason, **Center image** is cleared by default.

Another note of interest — some cameras pad their video output with zeros so that the display does not cut off the image on one side, usually the left. Images that include such padding may appear to be off-center on the display. In fact, while the displayed image may not appear centered, the electronic image (the data that compose the displayed image plus the padding which you cannot see) is centered in the display area.

See Also

DM642 EVM Video ADC

Purpose

Control LEDs

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

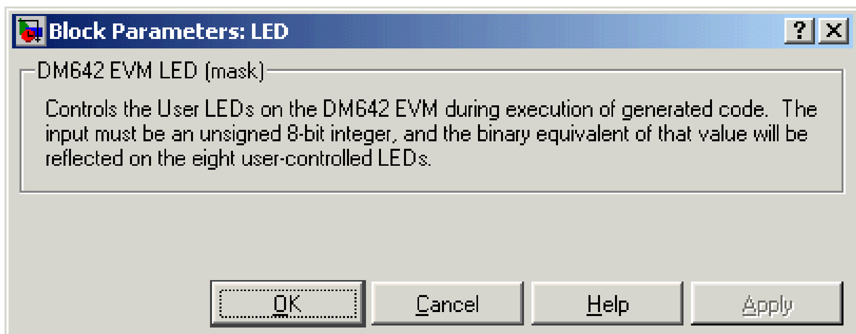
Description

LED

Controls the User LEDs on the DM642 EVM while the processor executes your generated code. To trigger the LEDs, input an unsigned 8-bit integer to the block. In response, the eight user-controlled LEDs reflect the binary equivalent of that input value — turning off an LED is 0 and turning on an LED is 1.

During operation, the LED block inherits the sample time from the upstream block in the model. Therefor, each time the model operation encounters the LED block, the block writes the desired output value to the LEDs.

Dialog Box



You see the block does not provide user options. Adding the block to your model adds the ability to control the LEDs.

DM642 EVM Video Port

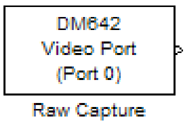
Purpose

Video port to receive video data from video input port

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description



Adding this block to your model lets you define the format of raw video captured by the video port on the DM642 EVM. The block outputs video as a stream of image frames built from the defined input.

You can select the video port the block reads from, set the size of the input data in bits per pixel, and define the frame sizes in pixels and lines.

When your process captures standard video input, like NTSC format video, another block for the DM642 EVM may be appropriate — the DM642 EVM Video ADC block.

By default, the block settings define NTSC format input video to capture — 640 pixels wide by 480 lines tall using 8 bits per pixel.

The block does not check your inputs to determine whether they form valid frames. You must be sure the values you assign work for you application.

The block does not support video capture from port 2 on the EVM.

Blanking intervals, both horizontal and vertical, represent the time needed for the scan to return to the starting point of the next line (the horizontal blanking period) or field or frame (the vertical blanking period).

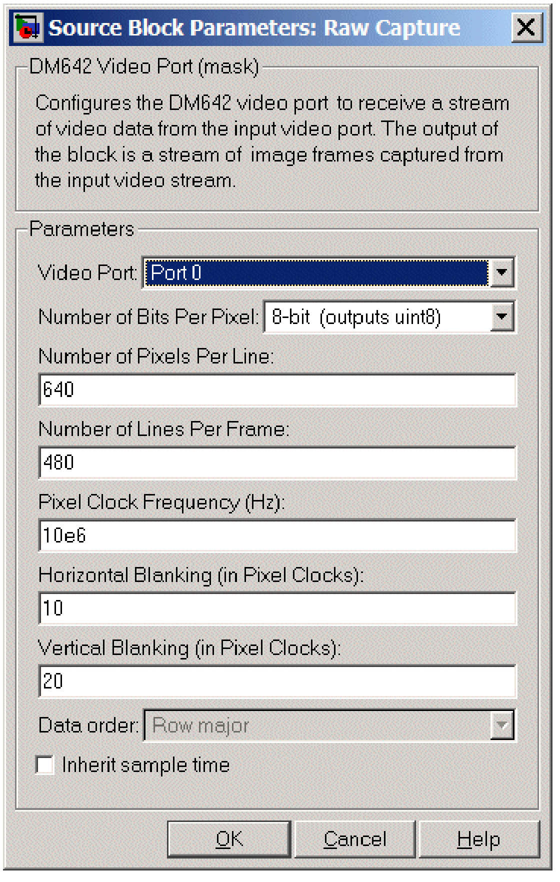
Memory Use

This block allocates video capture buffers on the system heap, using a TI driver that allocates three frame buffers on the heap for continuous video capture. To use the block you must create a heap in external memory on the target with the label EXTERNALHEAP. If you do not create the heap, either using the default values in the DM642 Target Preferences block or setting your own values. Embedded Coder software returns an error.

Use **Create heap** and **Heap size** and set the heap size in the DM642EVM Target Preferences block to configure the heap. Select **Define label** and name the heap EXTERNALHEAP in **Heap label**.

The default settings for the Target Preferences create a heap with sufficient memory to handle the worst case memory allocation needs automatically. If you configure the heap without sufficient memory, you get a run-time error because the system cannot initialize the video driver.

Dialog Box



Video Port

Select the video port to be the source of the raw video data stream. Either `0` or `1` appear on the list and `0` is the default port.

Number of bits per pixel

Select the number of bits used to represent a pixel in the input video stream. List entries tell you the input pixel representation and the data type of the output pixels for each input size. You cannot enter values here. Select from the list.

Number of pixels per line

Configure the width of each video frame in pixels. Enter the pixel count as an integer greater than zero.

Number of lines per frame

Configure the height of a single frame of video in lines. Enter the number of lines as an integer greater than zero. Combined with the **Number of bits per pixel**, this specifies the video frame format.

Pixel clock frequency

Specify the rate at which picture elements (pixels) arrive at the block input. Usually you enter this in Hz using scientific notation as shown by the default value. You can enter the value in decimal notation as well.

Horizontal blanking (in pixel clocks)

The blanking signal that occurs at the end of each video scanning line. Enter the value as an integer number of pixels. One video line comprises the number of pixels in the line plus the horizontal blanking pixels.

Vertical blanking (in pixel clocks)

The blanking signal that occurs at the end of each video field or frame. Enter this value as an integer number of lines (pixels). One frame includes the number of lines in the height of the frame plus the additional blanking lines.

Data order

With this option you tell the encoder whether to output video in row major or column major order. Most video capture and display systems use row major ordering. MATLAB and Simulink software use column major order. As a result, some Simulink blocks and MATLAB operations may not produce the output you expect unless you change the ordering for video from the default row major setting to column major.

Inherit sample time

Selects whether the block inherits the sample time from the model base rate or Simulink base rate as determined 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

DM642 EVM Video ADC, DM642 EVM Video DAC

Purpose

Reset to initial conditions

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM642 EVM

Description

Reset

Double-clicking this block in a Simulink model window resets the DM642 EVM that is running the executable code built from the model. When you double-click the Reset block, the block runs the software reset function provided by CCS IDE that resets the processor on your DM642 EVM. Applications running on the board stop and the signal processor returns to the initial conditions you defined.

Before you build and download your model, add the block to the model as a stand-alone block. You do not need to connect the block to any block in the model. When you double-click this block in the block library it resets your DM642 EVM. In other words, anytime you double-click a DM642 EVM Reset block you reset your DM642 EVM.

Dialog Box

This block does not have settable options and does not provide a user interface dialog box.

Purpose

Configure AIC33 audio codec to capture audio stream from LINE-IN or MIC

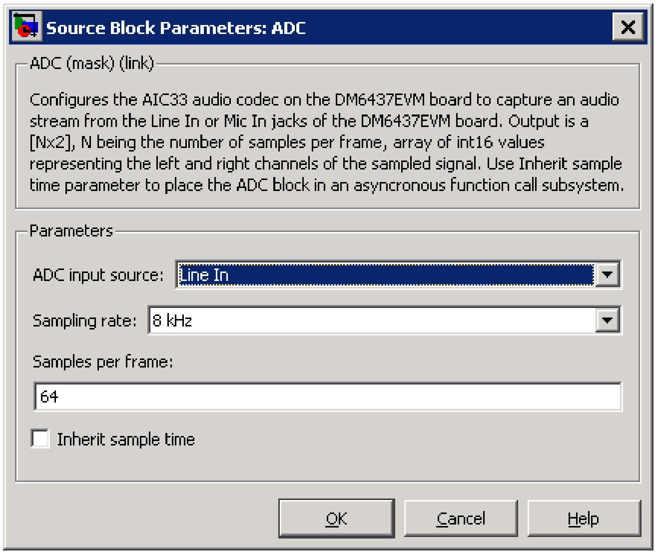
Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description

This block uses the AIC33 audio codec on the DM6437 EVM board to capture an analog audio stream from the **Line In** or **Mic** jacks and generate a digital frame-based output. Output is a [Nx2] array of int16 values representing the left and right channels of the sampled signal, where N is the number of samples per frame. Use the **Inherit sample time** parameter to place the ADC block in an asynchronous function call subsystem.

Dialog Box



ADC input source

Select **Line In** or **Mic In** as the input source.

Sampling Rate

Set the sampling rate of the analog-to-digital converter, from 8 kHz (the default) to 96 kHz.

Samples per frame

Set the number of samples the block 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. For example, if **Sampling Rate** is 8 kHz, and **Samples per frame** is 32, the 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 Simulink base rate as determined 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

DM6437 EVM DAC

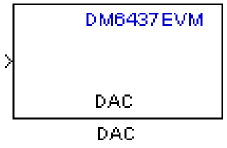
Purpose

Configure AIC33 codec to convert digital signal to audio output on LINE OUT and HP OUT

Library

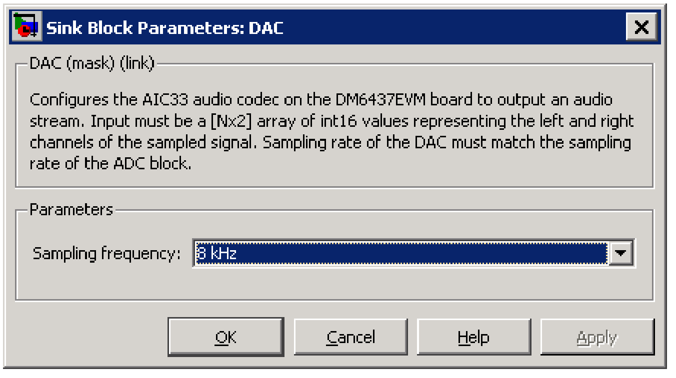
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description



Configure the AIC33 stereo codec on the DM6437 EVM board to convert a digital signal to an analog audio stream on the LINE OUT and HP OUT output jacks. The digital signal input must be an [Nx2] array of int16 values. Column 1 of the array is the left channel and column 2 is the right channel of the sampled signal. The sampling rate of the DAC output must match the sampling rate of the digital signal from the ADC.

Dialog Box



Sampling frequency
: Select the sampling rate of the digital signal input. This value must match the **Sampling rate** of the ADC block in your model.

See Also

DM6437 EVM ADC

DM6437 EVM DIP

Purpose

Output state of user-selected DIP switch as Boolean

Library

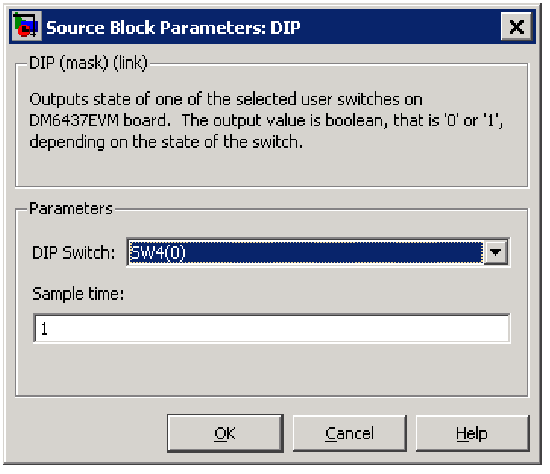
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description

DIP

Outputs the state of a user-selected DIP switch or jumper on the DM6437 EVM board. The output is a Boolean value, 0 (open) or 1 (closed). Use multiple blocks to output the state of multiple DIP switches.

Dialog Box



DIP Switch

Select the switch or jumper to sample: SW4(0),SW4(1), SW4(2), SW4(3), JP1, SW7.

SW4 is a read-only user switch. JP1 is for NTSC/PAL selection. SW7 is a slide switch.

Sample time

The interval between samples, in seconds. This value defaults to 1 second between samples.

DM6437 EVM LED

Purpose

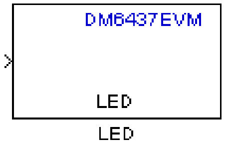
Apply Boolean input to user-selected LED

Library

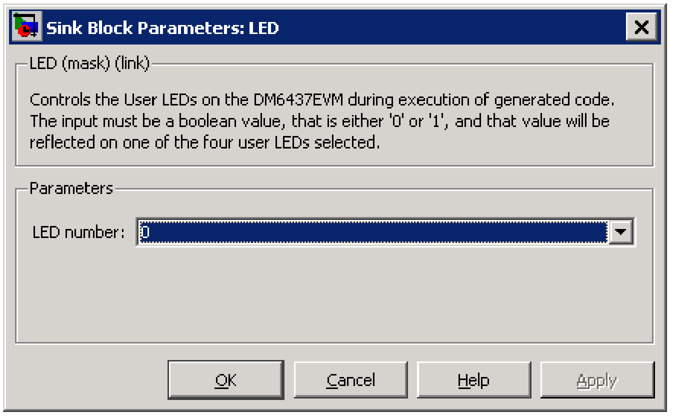
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description

This block controls an individual LED among the User LEDs on the DM6437 EVM during execution of generated code. The block input accepts Boolean values, 0 (off) or 1 (on). Use multiple blocks to control multiple LEDs.



Dialog Box



LED number

Specify the number of the User LED that the Boolean input controls.

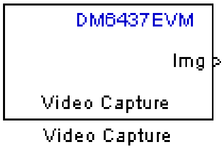
Purpose

Configure video peripherals to capture NTSC/PAL video

Library

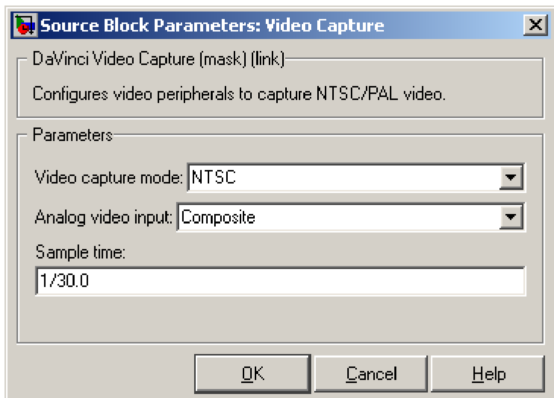
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description



Configure the video peripherals to capture an NTSC/PAL video input and make it available as a stream of YCbCr 4:2:2 interleaved data.

Dialog Box



Video capture mode

Set the video format to match that of the input, **NTSC** or **PAL**.

Analog video input

Set the input type to match that of the input, **Composite** or **S-video**.

Sample time

Set a sample time rate that matches the frame rate of the input signal, typically `1/30` for NTSC and `1/25` for PAL. A mismatch between these two rates may cause discontinuities in the video output signal.

See Also

DM643x Draw Rectangles, DM643x OSD, DM643x Video Display

Purpose

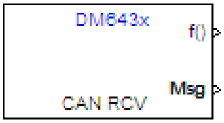
Receive messages from CAN serial communications bus on DM643x

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

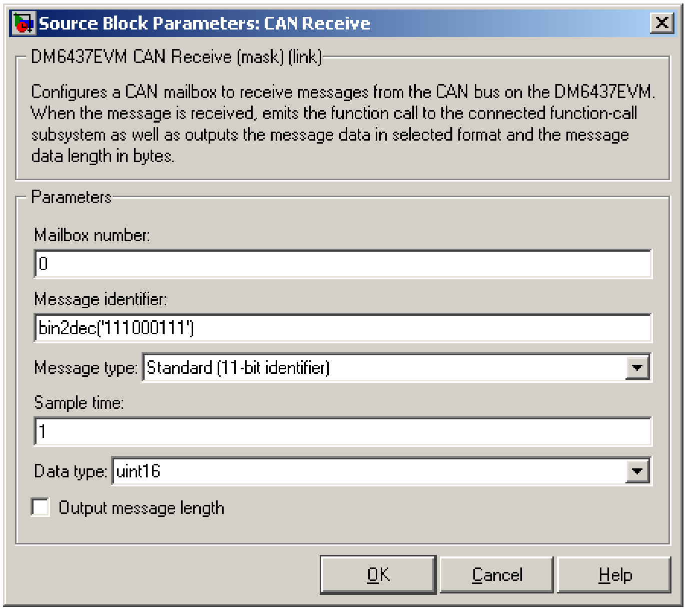
Description



CAN Receive

The CAN Receive block listens to broadcast messages on the DM643x CAN protocol bus. It saves messages with the user-specified **Message Identifier** to its message buffer. The CAN Receive block polls the message buffer at a rate determined by **Sample time**. When it detects a message in the message buffer, the block triggers the function-call output (f0) and makes the CAN message data available at the message output (Msg).

Dialog Box



Mailbox number
: Enter a unique number from `0` to `15` for standard or from `0` to `31` for enhanced CAN mode. This field refers to a mailbox area in RAM. In standard mode, the mailbox number determines priority.

Message identifier
: Identifies the length of the message—11 bits for standard frame size or 29 bits for extended frame size in decimal, binary, or hex formats. If the format is binary or hex, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry. The message identifier is associated with a receive mailbox. This mailbox only accepts messages that match the mailbox message identifier.

Message type
: Select `Standard (11-bit identifier)` or `Extended (29-bit identifier)`.

Sample time
: Frequency with which the mailbox is polled to determine if a new message has been received. A new message causes a function call to be emitted from the mailbox. To update the message output only when a new message arrives, the block must be executed asynchronously. To execute this block asynchronously, set **Sample Time** to `-1`. Refer to "Asynchronous Scheduling" for a discussion of block placement and other necessary settings.

For information about setting the timing parameters of the CAN module "Configuring Timing Parameters for CAN Blocks".

Data type
: Type of data in the data vector. The length of the vector for the received message is, at most, 8 bytes. If the message is less than 8 bytes, the data buffer bytes are right-aligned in the output. Only `uint16` (vector length = 4 elements) or `uint32` (vector length = 8 elements) data are allowed. This block uses an 8–byte data buffer to unpack the data, as follows:

For `uint16` data,

```
Output[0] = data_buffer[1..0];
Output[1] = data_buffer[3..2];
Output[2] = data_buffer[5..4];
Output[3] = data_buffer[7..6];
```

For uint32 data,

```
Output[0] = data_buffer[3..0];
Output[1] = data_buffer[7..4];
```

For example, if the received message has two bytes,

```
data_buffer[0] = 0x21
data_buffer[1] = 0x43
```

the uint16 output would be:

```
Output[0] = 0x4321
Output[1] = 0x0000
Output[2] = 0x0000
Output[3] = 0x0000
```

Output message length

Select this option to output the message length, in bytes, to the third output port. If you do not select this option, the block has only two output ports.

References

For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981, available at the Texas Instruments Web site.

See Also

"Configuring Timing Parameters for CAN Blocks", DM643x CAN Setup, DM643x CAN Transmit

DM643x CAN Setup

Purpose

Configure CAN serial communications bus parameters on DM643x

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

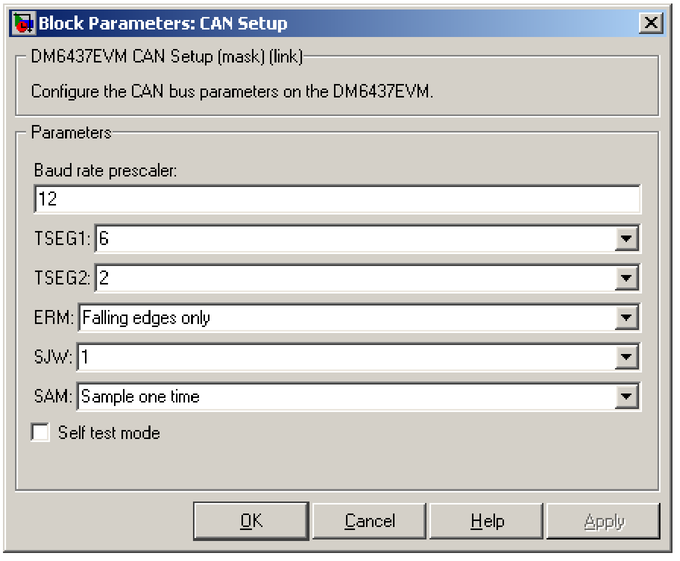
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description

CAN Setup

This block configures the CAN serial communications bus parameters on the DM6437EVM. The "Configuring Timing Parameters for CAN Blocks" topic provides instructions and examples for configuring this block.

Dialog Box



Baud rate prescaler
Value by which to scale the bit rate. Valid values are 0 to 255.

TSEG1

(Time SEGment 1) Sets the value of time segment 1, which, with **TSEG2** and **Baud rate prescaler**, determines the length of a bit on the CAN bus. Valid values for **TSEG1** are 2 through 16.

TSEG2

(Time SEGment 2) Sets the value of time segment 2, which, with **TSEG1** and **Baud rate prescaler**, determines the length of a bit on the CAN bus. Valid values for **TSEG2** are 2 through 8.

ERM

(Edge Resynchronization Mode) Sets the message resynchronization triggering. Options are `Falling edges only` and `Both falling and rising edges`.

SJW

(Synchronization Jump Width) For CAN to work successfully, all nodes on the network must be synchronized. However, as time passes, clocks on different nodes drift out of sync, and must resynchronize. **SJW** specifies the maximum width (in time quanta) that can be added to **TSEG1** (in the case of a slower transmitter), or subtracted from **TSEG2** (in the case of a faster transmitter) to regain synchronization during the receipt of a CAN message. Valid values for **SJW** are 1 to 4.

SAM

(SAMple point setting) 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 (Time Quanta/2). A majority decision is derived from the three points.

Self test mode

Puts the CAN module into loopback mode, that sends a dummy acknowledge message without requiring an acknowledge bit.

DM643x CAN Setup

References For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981, available at the Texas Instruments Web site.

See Also "Configuring Timing Parameters for CAN Blocks", DM643x CAN Transmit, DM643x CAN Receive

Purpose

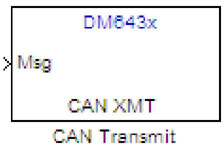
Configure CAN mailbox to transmit messages on CAN serial communications bus on DM643x

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

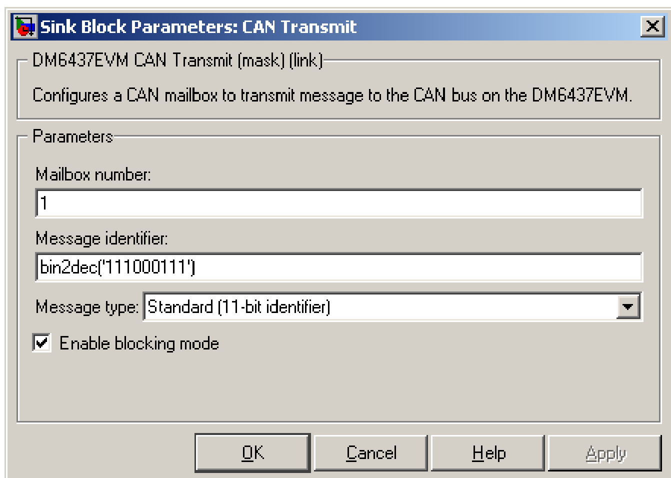
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



The CAN Transmit block receives messages through the message input (Msg) and broadcasts them to the CAN serial communication bus on the DM643x.

Dialog Box



Mailbox number

Sets the value of the mailbox number register (MBNR). For standard CAN controller (SCC) mode, enter a unique number from 0 to 15. For high-end CAN controller (HECC) mode enter a unique number from 0 to 31 . In SCC mode, transmissions from the mailbox with the highest number have the highest priority. In HECC mode, the mailbox number only determines priority if the Transmit priority level (TPL) of two mailboxes is equal.

Message identifier

Sets the value of the message identifier register (MID). The message identifier is 11 bits long for standard frame size or 29 bits long for extended frame size in decimal, binary, or hex format. For the binary and hex formats, use `bin2dec(' ')` or `hex2dec(' ')`, respectively, to convert the entry.

Message type

Select `Standard (11-bit identifier)` or `Extended (29-bit identifier)`.

Enable blocking mode

If you enable blocking mode, the CAN block code blocks further transmissions indefinitely until it receives a successful transmit acknowledge (TA bit in the CANTA register = 1). If you disable blocking mode, the CAN block code continues transmitting without receiving successful transmit acknowledgements. This is useful when the hardware might fail to acknowledge transmissions.

References

For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981, available at the Texas Instruments Web site.

See Also

"Configuring Timing Parameters for CAN Blocks", DM643x CAN Setup, DM643x CAN Receive

DM643x Draw Rectangles

Purpose

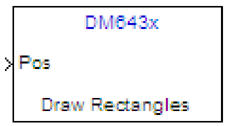
Configure Video Processing Back End to draw rectangles using On Screen Display (OSD) module

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

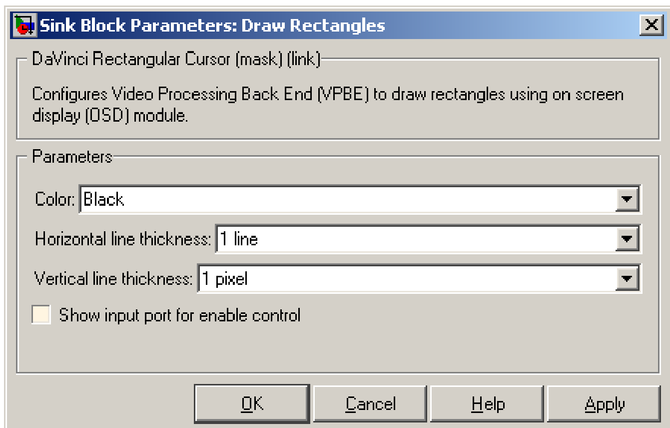
Description



Draw Rectangles

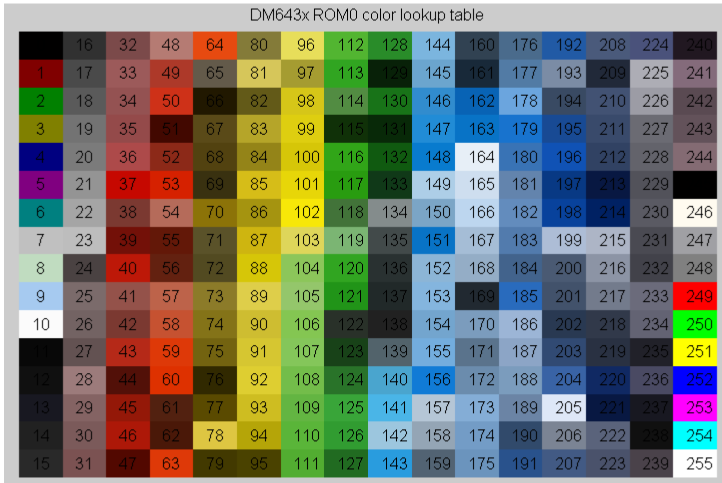
This block configures the Video Processing Back End (VPBE) to draw and position rectangles using the On Screen Display (OSD) module. The position input (**Pos**) is a 1x4 vector, designates the location of the upper-left corner of the rectangle. The position coordinates (`0,0`) originate in the upper-left corner of the video display.

Dialog Box



Color

Select the rectangle color. For **Specify via dialog**, enter an integer between 0–255. This integer specifies a corresponding RGB color in the DM643x ROM0 color lookup table (DM643x ROM0 CLUT). If you select **Specify via input port**, the block displays an additional input port, Color. Like **Specify via dialog**, the Color input takes an integer between 0–255 that fetches a color from the DM643x ROM0 CLUT. Changing the input value to the Color input port can change the color of the rectangle while the model is running.



For more information about the DM643x ROM0 CLUT, enter the following text at the MATLAB command prompt:

```
help 'dm643x_clut'
```

Horizontal line thickness
Select the cursor height in lines.

Vertical line thickness
Select the cursor width in pixels.

Show input port for enable control
Create an input port (**En**) that can be used to enable or disable the position input.

See Also

DM643x OSD, DM643x Video Capture, DM643x Video Display

Purpose

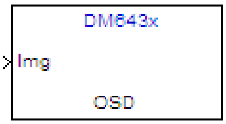
Overlay graphics and text on video

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description

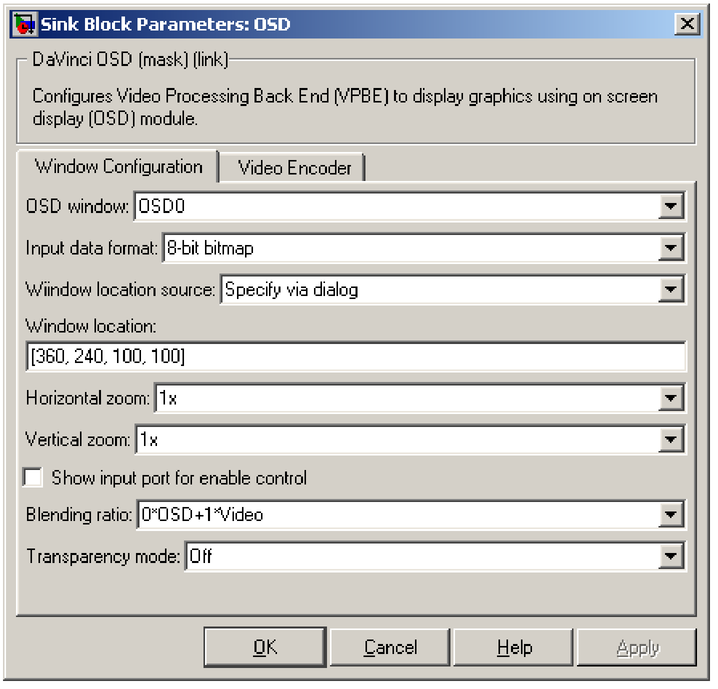


OSD

Use the On Screen Display (OSD) capabilities of the Video Processing Back End (VPBE) to overlay graphics and text on video.

Dialog Box

Window Configuration Pane



OSD window

Display graphics using OSD window 0 or 1.

Window Mode

If you set **OSD Window** to **OSD1**, the **Window Mode** parameter appears. Selecting **Display** configures OSD1 to display graphics. Selecting **Attribute** configures OSD1 to serve as an “alpha” input for controlling the transparency of OSD0. The positions of the two OSD windows must match for this to work.

Input data format

Set the format of the input data to 1-, 2-, 4-, 8-bit bitmap, or RGB565 which provides 16-bit color depth (64k colors).

Due to bandwidth constraints, RGB565 can only be used with one OSD window at a time. If you are using OSD1 to control transparency (i.e., OSD1 **Window Mode** is **Attribute**), get the best color depth by setting OSD1 **Input data format** to one of the bitmap settings and OSD0 **Input data format** to **RGB565**.

Window location source

Select the method for setting the location of the graphics display window. **Specify via dialog** creates the **Window location** field. **Specify via input port** creates an position input (Pos) on the OSD block which accepts the location of the window as data.

Window location

This parameter appears when you set **Window location source** to **Specify via dialog**. Set the pixel width, height, and base coordinates. For example, the default values, [360, 240, 100, 100] set the width to 360 pixels, the height to 240 pixels, the base coordinates for x to 100 pixels, and the base coordinates for y to 100 pixels.

Note [0, 0], the origin of the coordinate system, is the located in the upper-left corner of the Video0 window.

Horizontal zoom

Set the horizontal magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the OSD block.

Vertical zoom

Set the vertical magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the OSD block.

Show input port for enable control

Create an input port (**En**) to enable or disable the OSD graphics display window. This parameter is not available when **Window Mode** is **Attribute**.

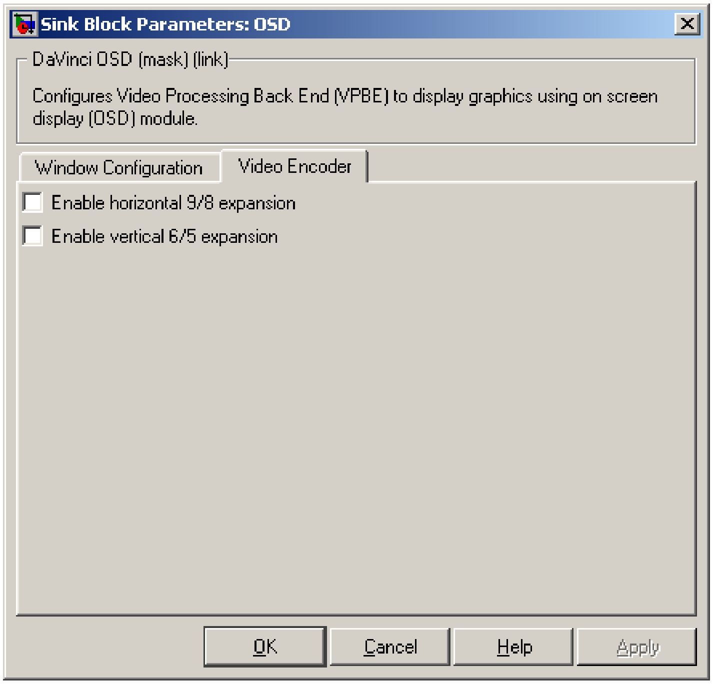
Blending ratio

Control the degree of blending between the OSD graphics display window and the Video display window in the background. This can be used to superimpose a semitransparent OSD graphic on a video background or to create fade-in and fade-out effects. The settings range from full OSD to full video in steps of 1/8. An additional setting, **Specify via input port**, creates an input port (**Blend**) for changing the ratio dynamically.

Transparency mode

Turn the transparency mode of the graphics display window **On** or **Off**, or select **Specify via input port** to create an input (**Trans**) on the OSD block. With transparency enabled, OSD pixels that match the color of the Video background color are rendered transparent. This is used for typical “bluescreen” type effects.

Video Encoder Pane



Enable horizontal 9/8 expansion
: Expands the image horizontally and is typically used to compensate for spatially compressed NTSC and PAL video signals. For example, you can use this setting to correct a 720 x 480 pixel NTSC analog video input that is displayed as a 640 x 480 pixel image.

Enable vertical 6/5 expansion
: Expands the image vertically and is typically used to compensate for spatially compressed PAL video signals. For example, you can use this setting in combination with the **Enable horizontal 9/8 expansion** setting to correct a 720 x 576 pixel PAL analog video input that is displayed as a 640 x 480 pixel image.

See Also DM643x Draw Rectangles, DM643x Video Capture, DM6437 EVM Video Capture, DM643x Video Display

Purpose

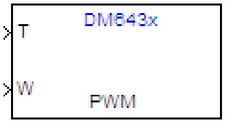
Configure DM643x DSP Event Manager to generate PWM waveforms

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



PWM

This block configures any one of the three PWM modules on the DM6437; each module has one output. The PWM module's clock cycles depend on the DM6437's 27 MHz input clock, and are not affected by the DM6437's PLL module. Upon startup, the PWM module uses the **Initial waveform period** and **Initial duty-cycle** values. Inputs to the waveform period port, **T**, and the duty-cycle port, **W**, can change those values while the application is running.

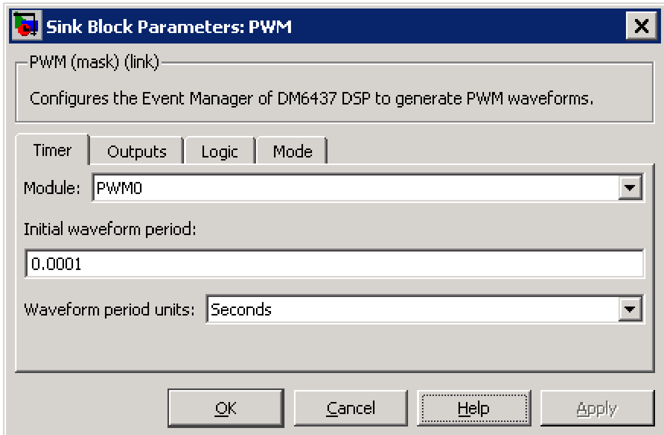
Dialog Box

The PWM block dialog box comprises four tabs:

- **Timer** — Select the PWM module, and configure the initial waveform.
- **Outputs** — Configure the initial duty cycle.
- **Logic** — Configure the control logic.
- **Mode** — Configure one-shot or continuous operation.

The following sections describe the contents of each tab in the dialog box.

Timer



Module

Select the PWM module for this block. All the parameter settings in this block configure the registers of the PWM module selected.

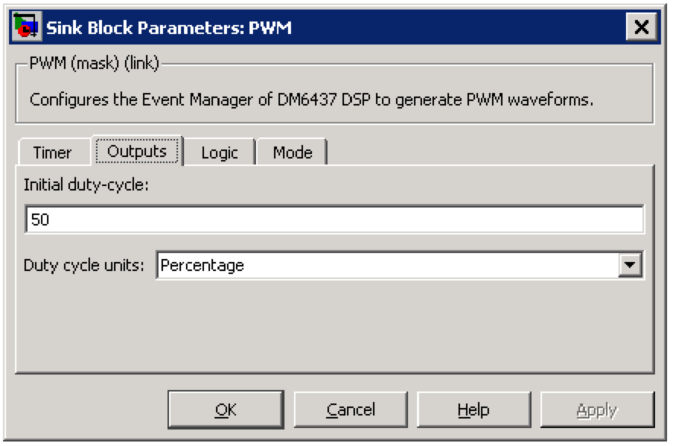
Initial waveform period

Set the initial period of the PWM waveform. The waveform period applied at the input port, **T**, changes this value. The range of acceptable values is 0.000000296 to 79.536431370 seconds or 8 to 2^{31}-1 clock cycles. These ranges depend on the 27 MHz clock frequency and the width of the 32-bit register.

Waveform period units

Set the unit of measure of the waveform period to **Seconds** or **Clock cycles**. This setting applies to both the **Initial waveform period** and the waveform period input, **T**. Clock cycles depend on the DM6437's 27 MHz input clock.

Outputs



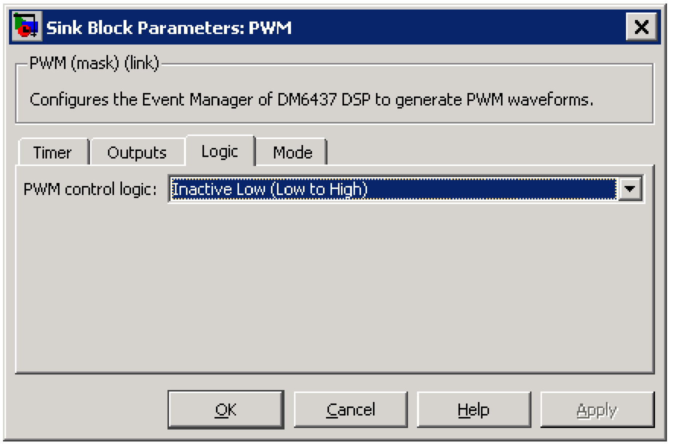
Initial duty-cycle

Set the initial duty-cycle of the PWM. The duty-cycle applied at the input port, **W**, changes this value. The range of acceptable values is 0 to 100 percent or 8 to 2^{31}-1 clock cycles. These ranges depend on the 27 MHz clock frequency and the width of the 32-bit register.

Duty-cycle units

Set the unit of measure of the duty-cycle to percentage or clock cycles. This setting applies to both the **Initial duty-cycle** and the duty-cycle input, **W**. Clock cycles depend on the DM6437's 27 MHz input clock.

Logic

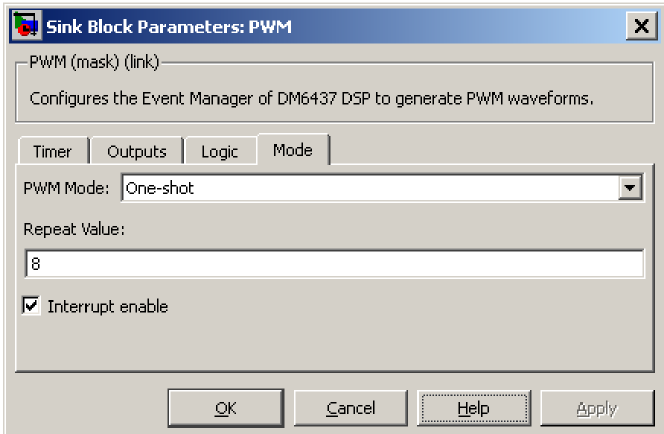


PWM control logic

Control the state of the PWM output while it is inactive and the polarity of the PWM waveform when it is active:

- **Inactive Low (Low to High)**: When the PWM output is inactive, the output remains low. When it is active, the first phase is low, and the second phase is high.
- **Inactive Low (High to Low)**: When the PWM output is inactive, the output remains low. When it is active, the first phase is high, and the second phase is low.
- **Inactive High (Low to High)**: When the PWM output is inactive, the output remains high. When it is active, the first phase is low, and the second phase is high.
- **Inactive High (High to Low)**: When the PWM output is inactive, the output remains high. When it is active, the first phase is high, and the second phase is low.

Mode



PWM Mode

Set the mode to one-shot or continuous. One-shot repeats the waveform for the number of periods given by repeat value and then, if interrupts are enabled, generates an interrupt at the end of operation. Continuous repeats the waveform infinitely and generates an interrupt, if enabled, every period.

Repeat Value

Set the repeat value if **PWM Mode** is set to **One-shot**. The PWM module outputs the waveform the specified number of times +1.

Interrupt enable

Enable the PWM module to generate an interrupt.

In one-shot mode, the PWM module generates an interrupt when number of periods given by **Repeat value** have been completed.

In continuous mode, the PWM module generates an interrupt during each period signaling that it is safe to set values for the subsequent waveform period and duty cycle.

References

For detailed information on the PWM module, see *TMS320DM643x DMP Pulse-Width Modulator (PWM) Peripheral User's Guide*, Literature Number SPRU995, available at the Texas Instruments Web site.

Purpose

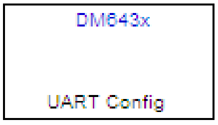
Configure DM643x UART for serial communication

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Description

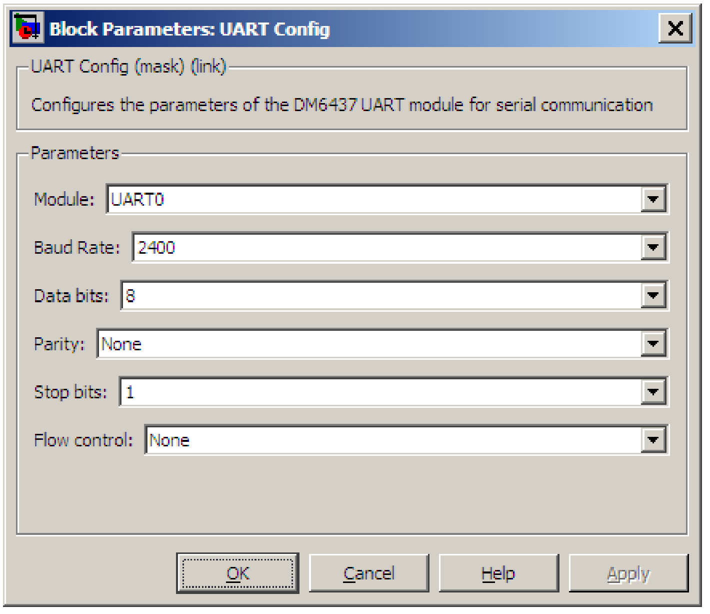


UART Config

Configure the serial communication parameters that are common to the transmit and receive elements of the DM643x UART module. If your model contains a DM643x UART Transmit block or a DM643x UART Receive block, it must also contain a DM643x UART Config block.

The UART module converts data between parallel and serial formats depending on whether it is transmitting or receiving data from external peripheral devices. Except for the **Module** parameter, configure all of the parameters in this block so they match the serial communication settings of the external peripheral devices.

Dialog Box



Module

Select the UART module this block configures, `UART0` or `UART1`. Your model can only contain one DM643x UART Config block per module.

Baud rate

Set the rate of signal modulations per second. Choose from `2400`, `4800`, `9600`, `19200`, `38400`, `57600`, or `115200`.

Data bits

Set the number of data bits in the character frame, from `5`, `6`, `7`, or `8`.

Parity

Enable and configure parity error detection.

In parity error detection, the transmitter reserves a parity bit at the end of the character frame, adds the number of 1's in the data bits, and assigns a value to the parity bit. The receiver compares the number of 1's in the data bits with the value of the parity bit. If the two values don't match, the receiver signals the transmitter that an error has occurred.

- **None** disables parity error detection. The character frame does not include a parity bit.
- **Odd** enables parity error detection and reserves a parity bit at the end of the character frame. If the data bits contain an odd number of 1's, the method assigns a value of 0 to the parity bit.
- **Even** enables parity error detection and reserves a parity bit to the end of the character frame. If the data bits contain an even number of 1's, the method assigns a value of 0 to the parity bit.

Stop bits
Select `1` or `2`.

Flow control
Select `None` or `Hardware`.

References

For detailed information on the UART module, see *TMS320DM643x DMP Universal Asynchronous Receiver/Transmitter (UART) User's Guide*, Literature Number: SPRU997, available at the Texas Instruments Web site.

See Also

DM643x UART Receive, DM643x UART Transmit

DM643x UART Receive

Purpose

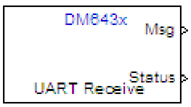
Configure receiver element of DM643x UART module for serial communication

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

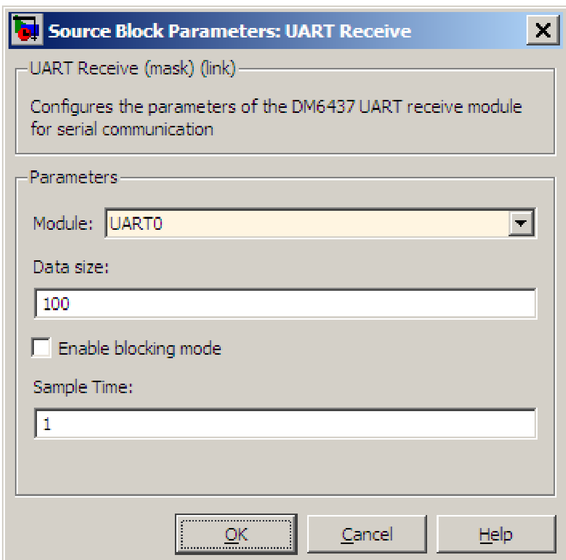
Description



UART Receive

Configure the serial communication parameters of the receiver element of the DM643x UART module. The receiver element converts data from external peripheral devices from serial to parallel format and passes it to the CPU. If your model contains a DM643x UART Receive block, it must also contain a DM643x UART Config block.

Dialog Box



Module

Select the UART module this block configures, `UART0` or `UART1`. Your model can only contain one DM643x UART Receive block per module. This parameter must also match the **Module** parameter in the DM643x UART Config block.

Data size

Set the data size, in bytes, of each transmission. Blocking mode uses this parameter to determine whether to generate an error.

Enable blocking mode

Enable this parameter to generate an error if the size of the last data transmission does not match the value of the **Data size** parameter. The DM643x UART Receive block sends the error message as a negative value on its **Status** output. If you disable **Enable blocking mode**, the block sends the number of bytes it received as a positive value on its **Status** output.

Sample time

Set the sample time for the block's input sampling. To execute this block asynchronously, set **Sample Time** to `-1`, and refer to "Asynchronous Scheduling" for a discussion of block placement and other necessary settings.

References

For detailed information on the UART module, see *TMS320DM643x DMP Universal Asynchronous Receiver/Transmitter (UART) User's Guide*, Literature Number: SPRU997, available at the Texas Instruments Web site.

See Also

DM643x UART Config, DM643x UART Transmit

DM643x UART Transmit

Purpose

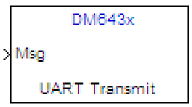
Configure transmitter element of DM643x UART module for serial communication

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

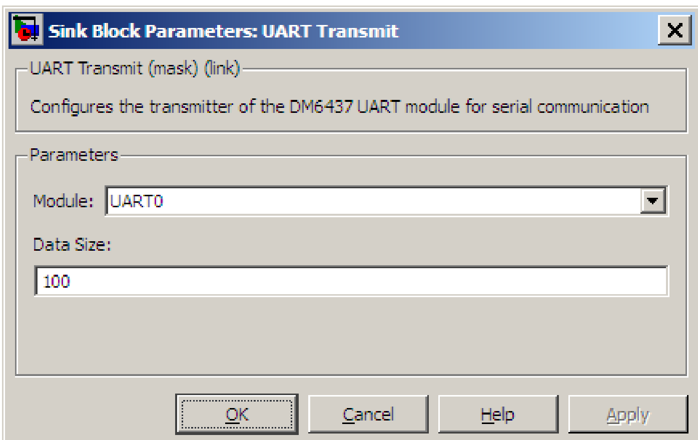
Description



UART Transmit

Configure the serial communication parameters of the transmitter element of the DM643x UART module. If your model contains a DM643x UART Receive block, it must also contain a DM643x UART Config block. The transmitter element converts parallel data from the CPU to a serial data format for output to external peripheral devices.

Dialog Box



Module

Select the UART module this block configures, `UART0` or `UART1`. Your model can only contain one DM643x UART Transmit block per module. This parameter must also match the Module parameter in the DM643x UART Config block.

Data size

Set the number of bytes to send per transmission.

References

For detailed information on the UART module, see *TMS320DM643x DMP Universal Asynchronous Receiver/Transmitter (UART) User's Guide*, Literature Number: SPRU997, available at the Texas Instruments Web site.

See Also

DM643x UART Config, DM643x UART Receive

DM643x Video Capture

Purpose

Configure Video Processing Front End (VPFE) to capture REC656 or generic YCbCr 4:2:2 video

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

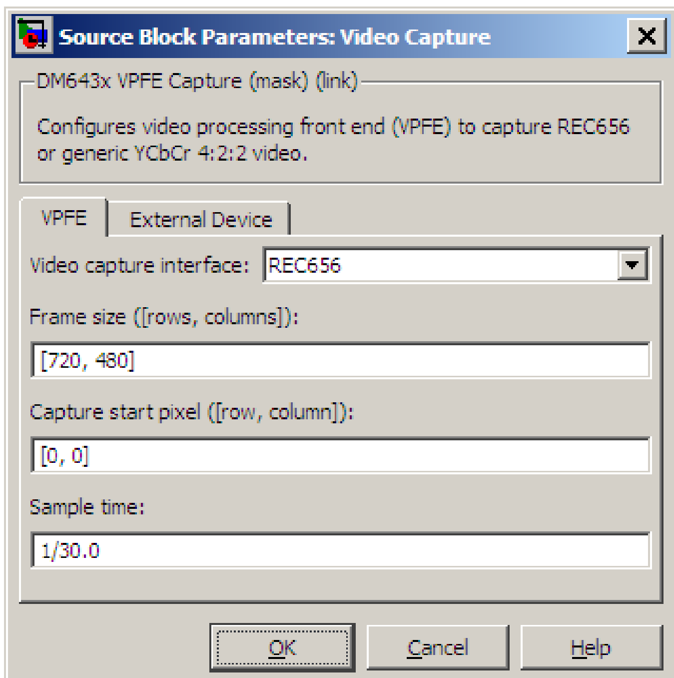
Description

Video Capture

Configure the video processing front end (VPFE) to capture NTSC or PAL video.

Dialog Box

VPFE



Video capture interface

Configure this parameter to match the format of the input signal using either the **REC656** or **Generic YCbCr-4:2:2** option. The **REC656** format is also known to as ITU-R BT.656 or CCIR-656 and comprises an 8-bit YCbCr 422 input signal. **Generic YCbCr-4:2:2** comprises an 8-bit signal with discrete horizontal (H) and vertical (VSYNC) signals, such as a computer monitor signal.

Data input mode

When **Video capture interface** is set to **Generic YCbCr-4:2:2**, set this parameter depending on the number of pins used by the physical interface. If the physical interfaces uses pins 0–8, select **8-bit**. If the physical interface uses pins 0–15, use **16-bit**. When you select **16-bit**, the lower 8 pins capture Y and the upper 8 pins capture the C (chroma) components.

For more information, refer to *Table 1. Interface Signals for Video Processing Front End* in the *TMS320DM643x DMP Video Processing Front End (VPFE) User's Guide*, Literature Number: SPRU977, available on the Texas Instruments Web site.

Scan mode

If you set **Video capture interface** to **Generic YCbCr-4:2:2**, set **Scan mode** to match the scan mode of the input signal, **Interlaced** or **Progressive**. Regardless of the setting, the block outputs an interleaved YCbCr 422 signal, which you can deinterleave using the C6000 Deinterleave block.

Note If you set **Scan mode** to `Interlaced`, verify that the Field ID signal is connected to the correct input pin for this video capture driver to work correctly.

Frame size

Define the size of the capture frame. You can use this parameter to capture the entire input frame or to capture just a portion of it.

The **Frame size** parameter values must be greater than zero and no greater than the size of the input frame. Enter the row and column dimensions of the capture frame in pixels. For example, entering `[740, 480]` sets the row width to 740 pixels, and the column height to 480 pixels.

Capture start pixel

Set the location of the capture frame relative to the display frame, using the upper-left corners of both frames (e.g., [0, 0]) as the point of reference. You can position the start pixel anywhere in the input frame. Enter the row and column dimensions of the Capture start pixel in pixels. For example, entering `[10, 20]` positions the upper-left corner of the capture frame at row 10, column 20 from the upper-left corner of the display frame.

The combination of the **Frame size** and **Capture start pixel** parameters may place the capture frame outside the display frame. If so, the portions of the capture frame that lie outside the display frame capture null video data (black screen) without generating an error.

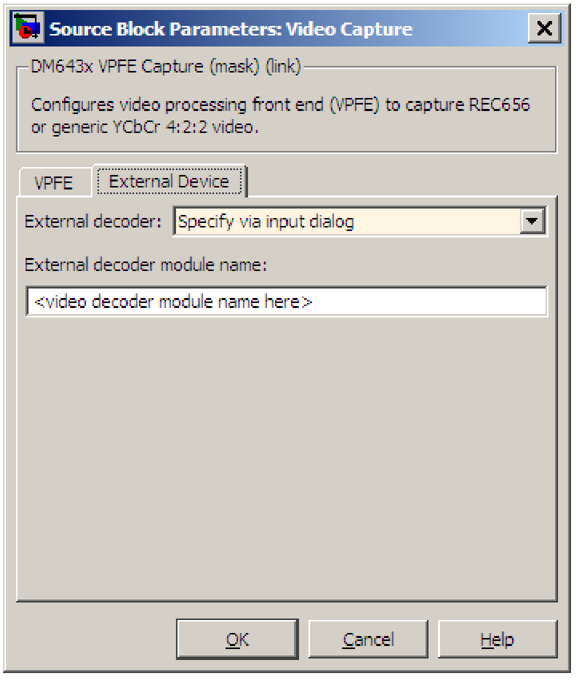
Sample time

Set the sampling rate of the video capture frame. Enter **Sample time** as a fraction of 1 over the sample rate per second. For example, to obtain a sample rate of 30 frames per second, enter `1/30.0`. NTSC has a typical frame rate of 1/30, while PAL usually requires 1/25.

You can set this parameter to match the frame rate of the input signal, or you can use it to downsample the input signal. For example, sampling a `1/30` input at `1/15` halves the data throughput of the signal.

Setting the sample time to a different value from the input signal refresh rate may cause discontinuities in the video image. Avoid exceeding the sample rate of the input signal.

External Device



The **External Device** tab enables you to connect a video device with an external video decoder to the VPFE. When you specify the external coder, you create hookpoints in the VPFE driver initialization code for opening the external video decoder, starting the data output, and closing the external video decoder. The external decoder plugs into the following function pointers:

- EVD_Handle (*Open)()
- Int (*Close)(Ptr handle)
- Int (*Control)(Ptr handle,Uint32 Cmd,Ptr CmdArg)

For example, if you were to enter "PSP_VPFE_TVP5146" for **External decoder module name**, you would declare the following functions as shown:

```
// External device open function
EVD_Handle PSP_VPFE_TVP5146_Open(void);
// External device close function
Int PSP_VPFE_TVP5146_Close(EVD_Handle handle);
// External device control function
Int PSP_VPFE_TVP5146_Control(EVD_Handle handle, Uint32 Cmd, Ptr CmdArg);
```

The VPFE driver also assumes that a user structure named TVP5146_ConfigParams and a variable called PSP_VPFE_TVP5146_params exists to pass to the PSP_VPFE_TVP5146_Control function. In other words, there must be a declaration like the following:

```
typedef struct _PSP_VPFE_TVP5146_ConfigParams
{
int dummy; // User defined fields
} PSP_VPFE_TVP5146_ConfigParams;
TVP5146_ConfigParams PSP_VPFE_TVP5146_params;
```

You must use the custom code interface to add the header file that declares function prototypes and the source files that contain the implementation of the _Open, *_Close and *_Control functions to the generated project. To see an example, download the Avnet S3ADSP DaVinci Evaluation Platform Support Package from http://www.mathworks.com/matlabcentral/fileexchange/22191, and open the model, avnet_test_dm6437evm.mdl. (Do not install the Avnet S3ADSP DaVinci Evaluation Platform Support Package. It is for R2008a only.)

External decoder

If your target is connected to a video device that outputs a RAW video signal and relies on the DM643x VPFE's built-in decoder, select **None**. If your target is connected to a video device with a decoder that outputs REC656 or generic YCbCr-4:2:2, select **Specify via input dialog**.

External decoder module name
If you set the **External decoder** to `Specify via input dialog`, then enter a name for the external video decoder module name in this field.

See Also

DM643x Draw Rectangles, DM643x OSD, DM643x Video Display

References

TMS320DM643x DMP Video Processing Front End (VPFE) User's Guide, Literature Number: SPRU977, available from the Texas Instruments Web site.

DM643x Video Display

Purpose

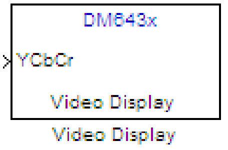
Configure Video Processing Back End to display NTSC/PAL video

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM6437 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



This block configures the Video Processing Back End (VPBE) to display NTSC/PAL video.

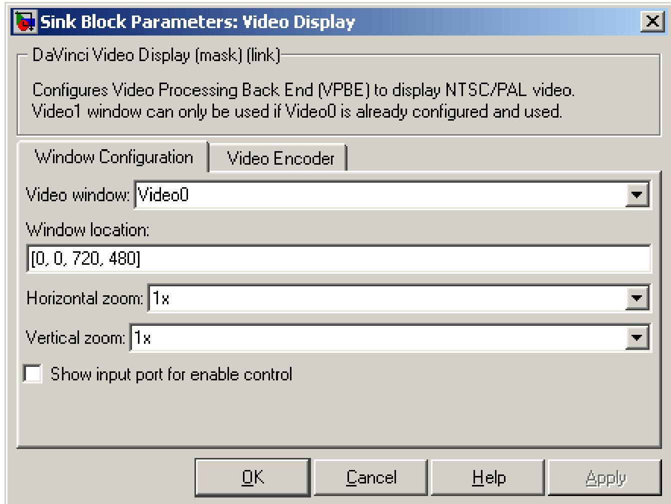
Dialog Box

The block dialog box comprises multiple tabs:

- **Window Configuration** — Configure the video window, position, zoom, and whether to display the input port.
- **Video Encoder** — Configure the video display mode, analog video output, and horizontal or vertical expansion.

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.

Window Configuration



Video window

Create a video display window, **Video0** or **Video1**.

You must create a **Video0** display window before you can use the following video elements:

- a Video1 video display window from the DM643x Video Display block
- an on-screen display from the DM643x OSD block
- a video rectangle from the DM643x Draw Rectangles block

Window location source

Select the method for setting the location of the graphics display window. **Specify via dialog** creates the **Window location** field. **Specify via input port** creates an position input (Pos) on the OSD block which accepts the location of the window as data.

Window location

This parameter appears when you set **Window location source** to **Specify via dialog**. Set the pixel width, height, and base coordinates. For example, the default values, [360, 240, 100, 100] set the width to 360 pixels, the height to 240 pixels, the base coordinates for x to 100 pixels, and the base coordinates for y to 100 pixels.

Note [0, 0], the origin of the coordinate system, is the located in the upper-left corner of the Video0 window.

Horizontal zoom

Set the horizontal magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the video display block.

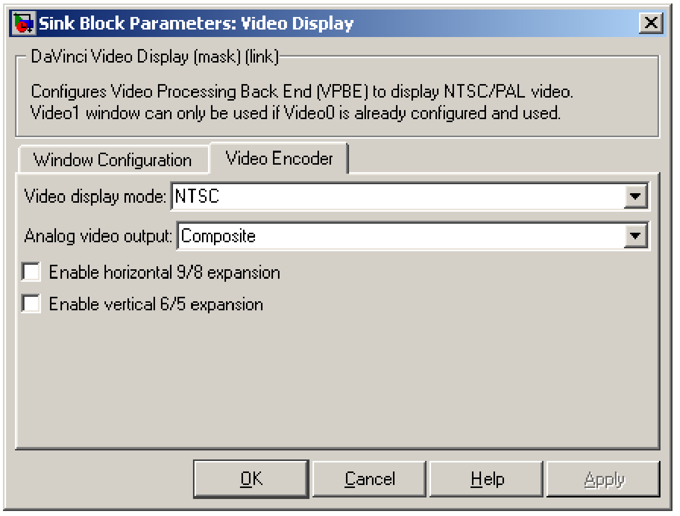
Vertical zoom

Set the vertical magnification of the graphics display window. Selecting **Specify via input port** creates a zoom input (**Zoom**) on the video display block.

Show input port for enable control

Create an input port (**En**) to enable or disable the video display window.

Video Encoder Pane



Video output mode

Set the output mode to **Analog** or **Digital**. This parameter is only available in the block that comes from the Avnet S3 ADSP DM6437 library.

Video display mode

Set the video format to **NTSC** , **PAL**, **HD 480p60**, or **HD 576p50**.

Analog video output

Set the output type to **Composite**, **S-video**, or **Component**.

Enable horizontal 9/8 expansion

Expands the image horizontally. Typically used to compensate for spatially compressed NTSC and PAL video signals. For example, use this setting to correct a 720 x 480 pixel NTSC analog video input that is displayed as a 640 x 480 pixel image.

Enable vertical 6/5 expansion

Expands the image vertically. Typically used to compensate for spatially compressed PAL video signals. For example, use this setting in combination with the **Enable horizontal 9/8 expansion** setting to correct a 720 x 576 pixel PAL analog video input that is displayed as a 640 x 480 pixel image.

See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

Purpose

Configure DSP peripherals to capture NTSC/PAL or HD video

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM648 EVM

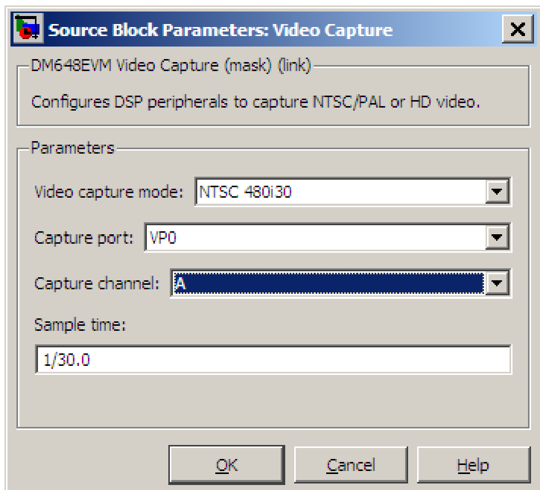
Description

Video Capture

This block configures the Video Processing Back End (VPBE) to capture NTSC, PAL, or HD video.

To capture multiple video data streams for applications such as multipicture displays, use multiple **Video capture** blocks. For NTSC and PAL, you can capture eight video streams by combining four **Capture ports** with two **Capture channels**. For HD, you can capture two video streams using two **Capture ports**.

Dialog



Video capture mode

Set the video format to **NTSC**, **PAL**, or **HD**. Each menu item gives the encoding type, the vertical lines of resolution, whether the scanning type is interlaced (i) or progressive (p), and the frame rate of the input. For example, the “NTSC 480i30” indicates

NTSC encoding, 480 lines of vertical resolution, interlaced, and 30 frames per second.

Capture port

Select the video input port. When you configure Video capture mode for an NTSC or PAL input, four capture ports become available. When you configure Video capture mode for an HD input, two capture ports become available. VP1 is not available in the list of capture ports because it is reserved for video display.

Capture channel

Two capture channels, A and B, are available for NTSC or PAL. **Capture channel** is not available when **Video capture mode** is configured for an HD input.

Sample time

Set the interval between samples in fractions of a second. This value defaults to 1/30.0, or one-thirtieth of a second. If the sample time does not match the frame rate of the video input, some irregularities may occur.

See Also

DM648 EVM Video Display

Purpose

Configure DSP peripherals to display NTSC, PAL, HD, or VESA video

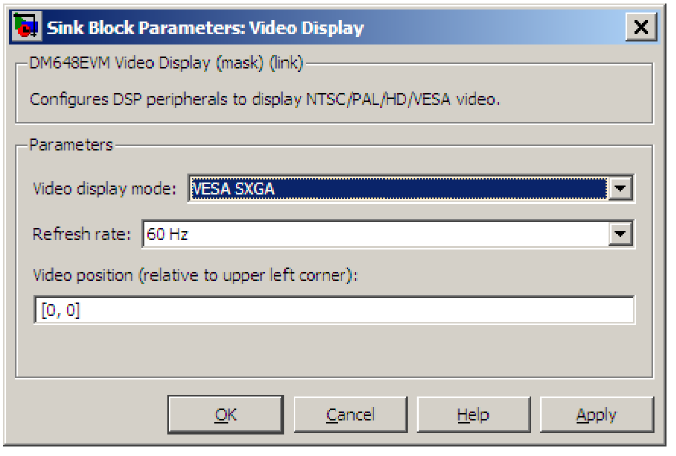
Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DM648 EVM

Description

This block configures the Video Processing Back End (VPBE) to display NTSC/PAL/HD/VESA video. When sending the video output to a computer display, verify that the combination of the resolution of the **VESA** in **Video display mode** and the frequency in **Refresh rate** are valid settings for the monitor. Using unsupported combinations may permanently damage the computer display connected to a video output.

Dialog Box



Video display mode

Set the video display mode to **NTSC**, **PAL**, **HD**, or **VESA**. The **NTSC**, **PAL**, and **HD** menu items give the encoding type, the vertical lines of resolution, whether the scanning type is interlaced (i) or progressive (p), and the frame rate of the input. For example, the "NTSC 480i30" indicates NTSC encoding, 480 lines of vertical resolution, interlaced, and 30 frames per second. The **VESA** modes correspond to a range of standard computer display modes.

Refresh rate

When **Video display mode** is one of the VESA modes, set the refresh rate of the video output.

Video position

Position the upper-left corner of the video output in the video display by entering coordinates. The default coordinates, `[0,0]`, correspond to the upper-left corner of the video display. Increasing the horizontal and vertical coordinates moves the video output to the right and down.

See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

Purpose

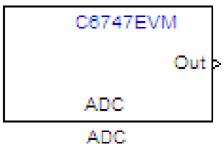
Capture audio stream from LINE IN jack

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6747 EVM

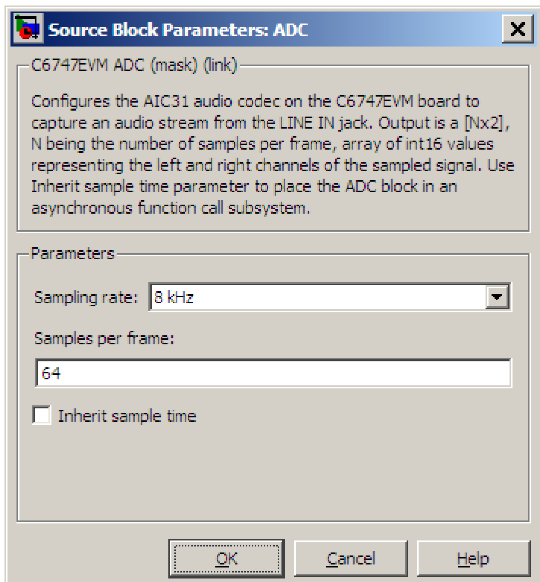
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6748 EVM

Description



Configures the AIC31 audio codec on the C6747EVM/C6748EVM board to capture an audio stream from the LINE IN jack. Output is a [Nx2], N being the number of samples per frame, array of int16 values representing the left and right channels of the sampled signal. Use Inherit sample time parameter to place the ADC block in an asynchronous function call subsystem.

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.

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 64, the resulting frame rate is 125 frames per second (8000/64 = 125).

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.

See Also

C6747 EVM/C6748 EVM DAC

Purpose

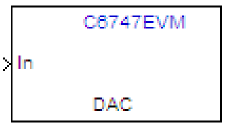
Output audio on LINE OUT / HP OUT jacks

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6747 EVM

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6748 EVM

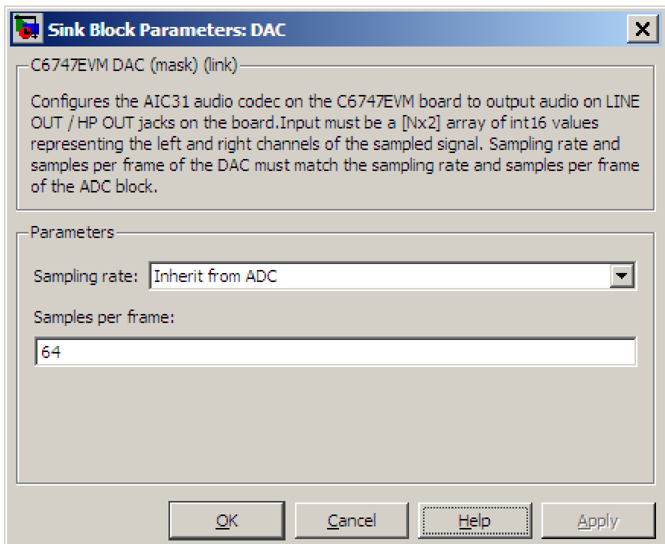
Description



DAC

Configures the AIC31 audio codec on the C6747EVM/C6748EVM board to output audio on LINE OUT / HP OUT jacks on the board.Input must be a [Nx2] array of int16 values representing the left and right channels of the sampled signal. Sampling rate and samples per frame of the DAC must match the sampling rate and samples per frame of the ADC block.

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, select `Inherit from ADC`.

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

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

Purpose

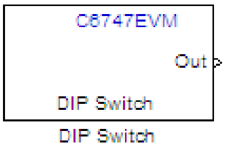
Output DIP switch status

Library

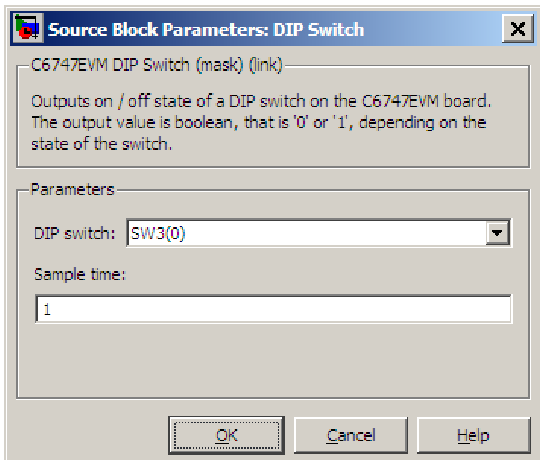
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6747 EVM

Description

Outputs on / off state of a DIP switch on the C6747EVM board. The output value is boolean, that is ’0’ or ’1’, depending on the state of the switch.



Dialog Box



DIP Switch
: Select the switch, 0 through 3, from the SW3 bank of switches.

Sample time
: Specify the time between samples of the signal in seconds. This value defaults to 1 second between samples.

See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

C6747 EVM LED

Purpose

Control four on-board LEDs

Library

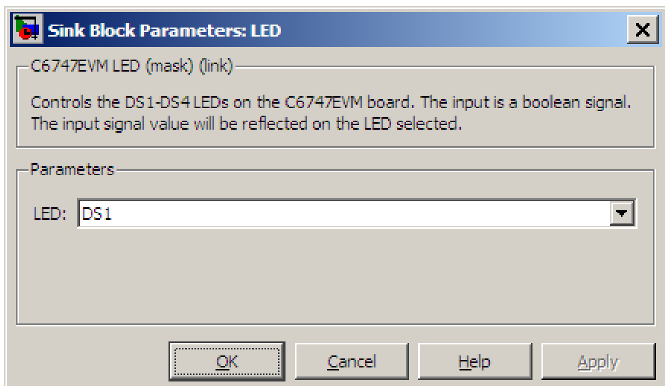
Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ C6747 EVM

Description

LED

Controls the DS1-DS4 LEDs on the C6747EVM board. The input is a boolean signal. The input signal value will be reflected on the LED selected.

Dialog Box



LED

Specify the number of the User LED that the Boolean input controls.

See Also

DM643x Draw Rectangles, DM643x OSD, DM6437 EVM Video Capture, DM643x Video Capture

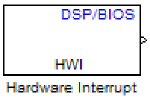
Purpose

Generate Interrupt Service Routine

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ 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.

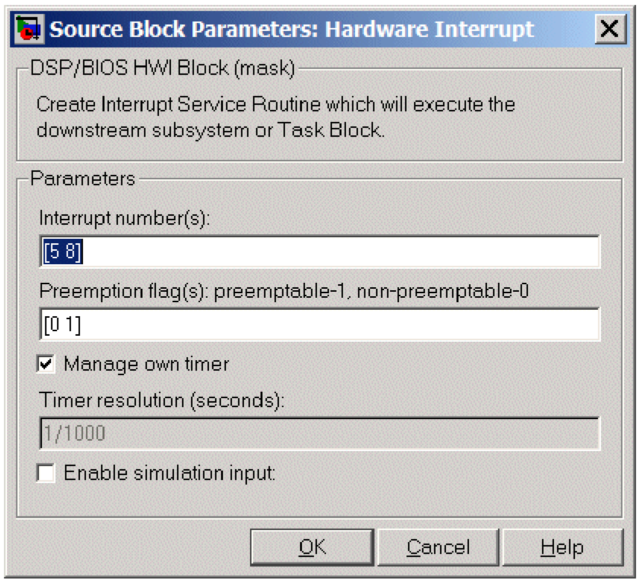
Interrupt numbers for C6000 family processors range from 0 to 15, with 0 reserved for the reset ISR. The following table presents the set of interrupt numbers for the C6713 processor. For more detailed and specific information about interrupts, refer to Texas Instruments technical documentation for your 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
9	EMUDTDMA	Emulation
10	SDINT	EMIF
11	EMURTDXRX	Emulation
12	EMURTDXTX	Emulation

Interrupt Number	Default Event	Module
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 any 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 affect generated code.

See Also

DSP/BIOS Task, DSP/BIOS Triggered Task

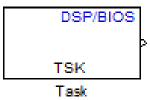
Purpose

Create task that runs as separate DSP/BIOS thread

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DSP/BIOS

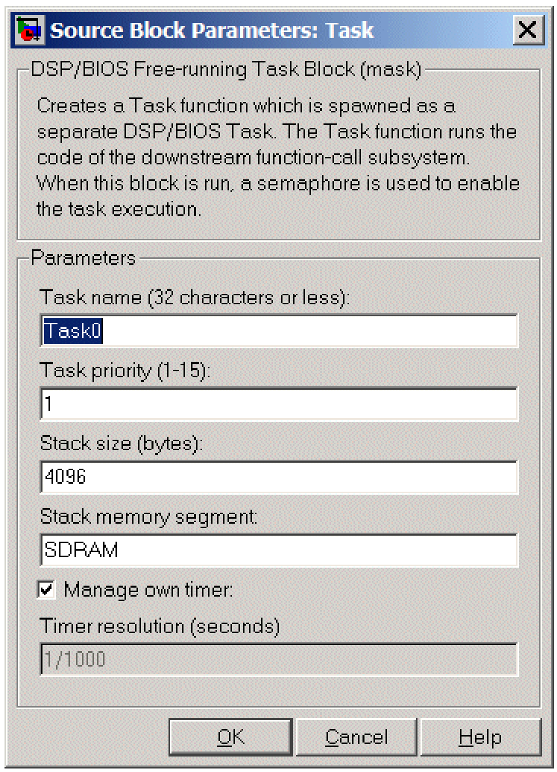
Description



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.

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 as needed. 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

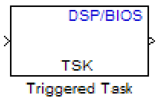
Purpose

Create asynchronously triggered task

Library

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ DSP/BIOS

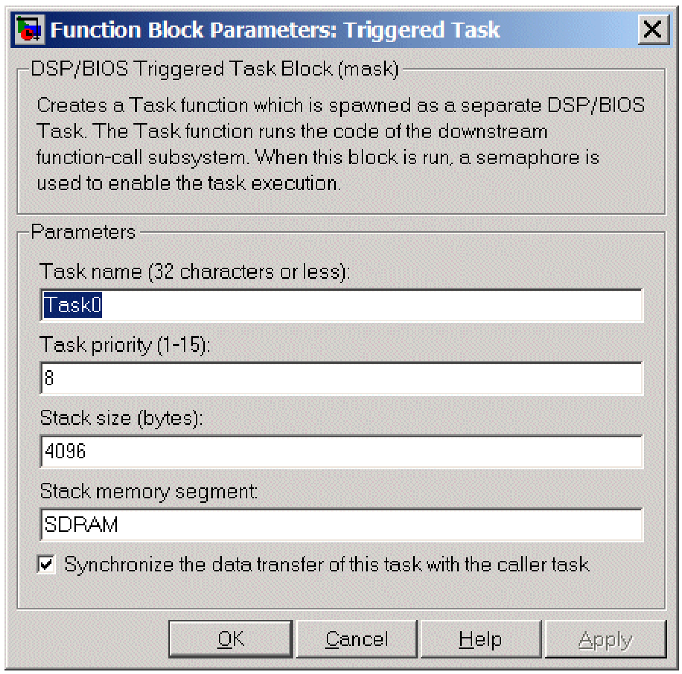
Description



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 as needed. 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 the stack size as large as necessary. 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 Preferences block in the model.

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

CAN Pack

Purpose

Pack individual signals into CAN message

Library

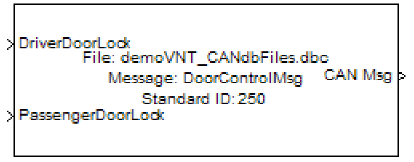
CAN Communication

Embedded Coder/ Embedded Targets/ Host Communication

Description

CAN Pack
(With raw data input)

CAN Pack
(With manually specified data input)



CAN Pack
(With CANdb specified data input)

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 input ports 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 input ports.

CAN Pack

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:

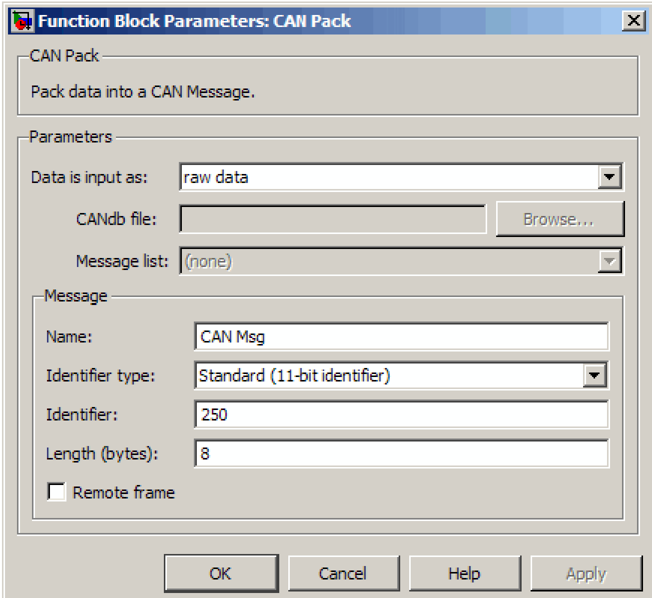
- The use of Simulink® Accelerator™ mode. Using this feature, you can speed up the execution of Simulink models.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Simulink Coder to deploy models to targets.

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.

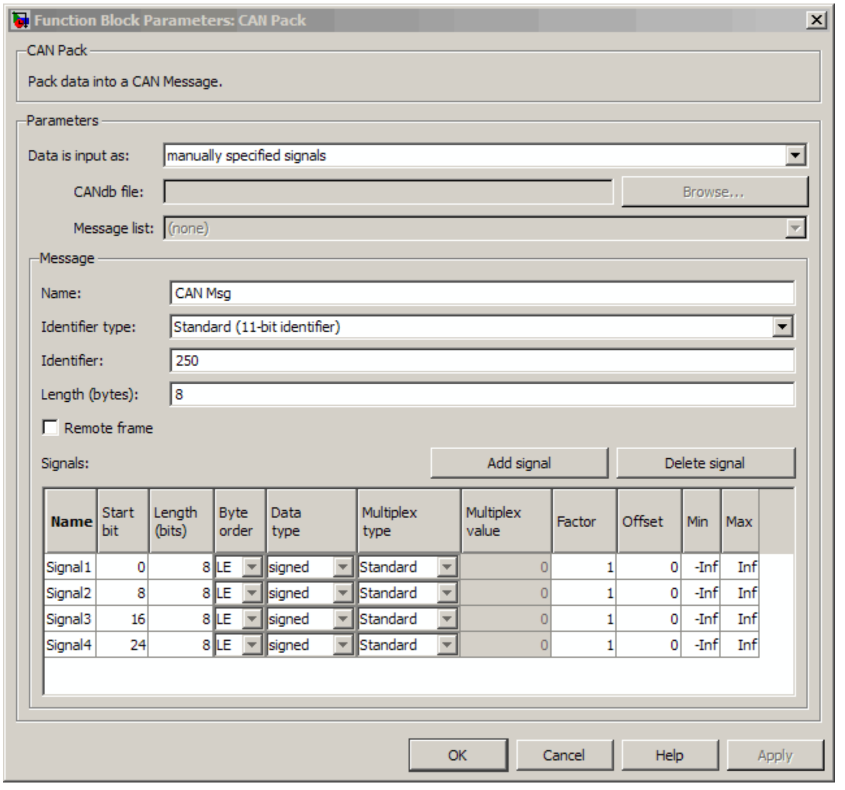


Parameters

Data is input as

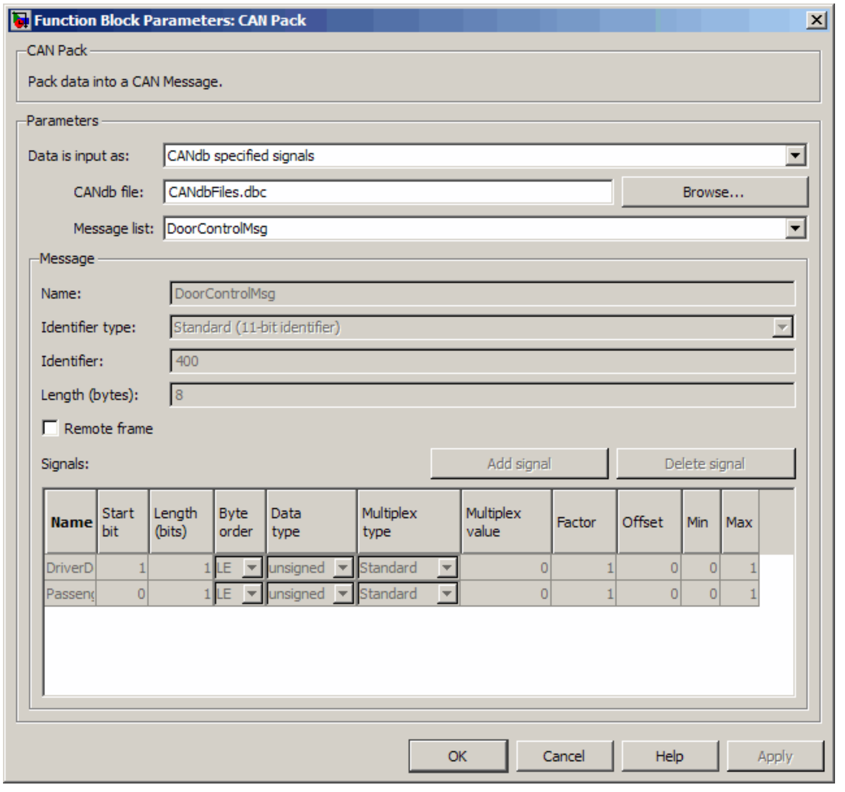
Select your data signal:

- **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 input ports on your block depends on the number of signals you specify.



- **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 input ports on your block depends on the number of signals specified in the CANdb file for the selected message.

Note You can specify a CAN database file only on a Windows 32–bit machine.



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 appropriate 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 any 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 in 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 any 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:

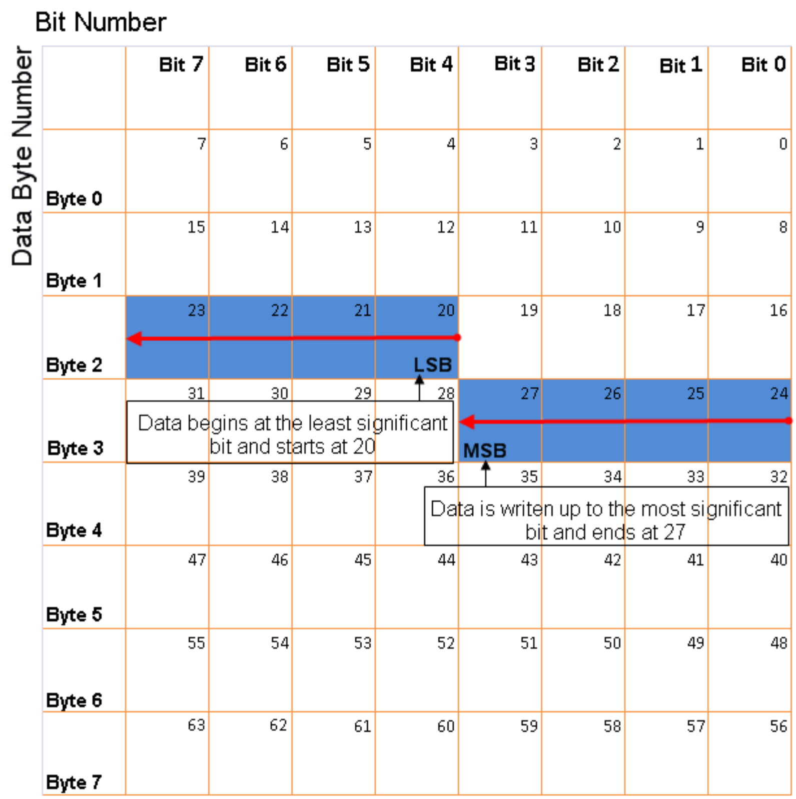
Name
Specify a descriptive name for your signal. The Simulink block in your model displays this name. The default is `Signal [row number]`.

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.

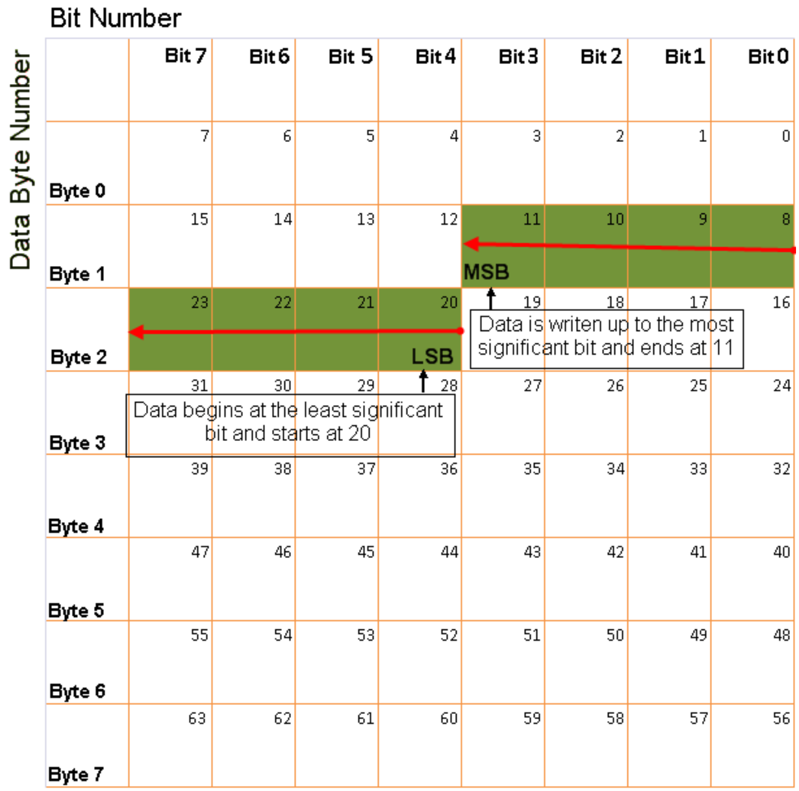
Byte order
Select either of the following options:

- `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.



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.



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 always packed at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is always 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 5-638 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 5-638 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 any number for the minimum value. See "Conversion Formula" on page 5-638 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 any number for the maximum value. See "Conversion Formula" on page 5-638 to understand how physical values are converted to raw values packed into a message.

Conversion Formula

The conversion formula is

```
raw_value = (physical_value - Offset) / 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

Purpose

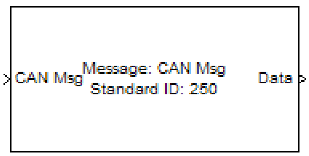
Unpack individual signals from CAN messages

Library

CAN Communication

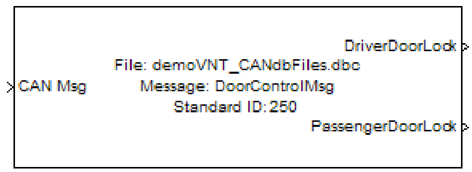
Embedded Coder/ Embedded Targets/ Host Communication

Description



CAN Unpack
(With raw data output)

CAN Unpack
(With manually specified data output)



CAN Unpack
(With CANdb specified data output)

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.

CAN Unpack

Other Supported Features

The CAN Unpack block supports:

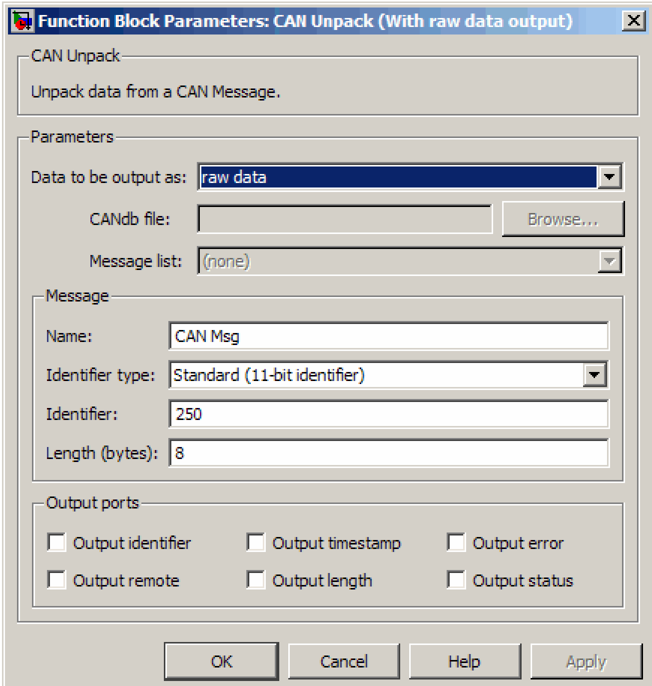
- The use of Simulink Accelerator mode. Using this feature, you can speed up the execution of Simulink models.
- The use of model referencing. Using this feature, your model can include other Simulink models as modular components.
- Code generation using Simulink Coder to deploy models to targets.

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.

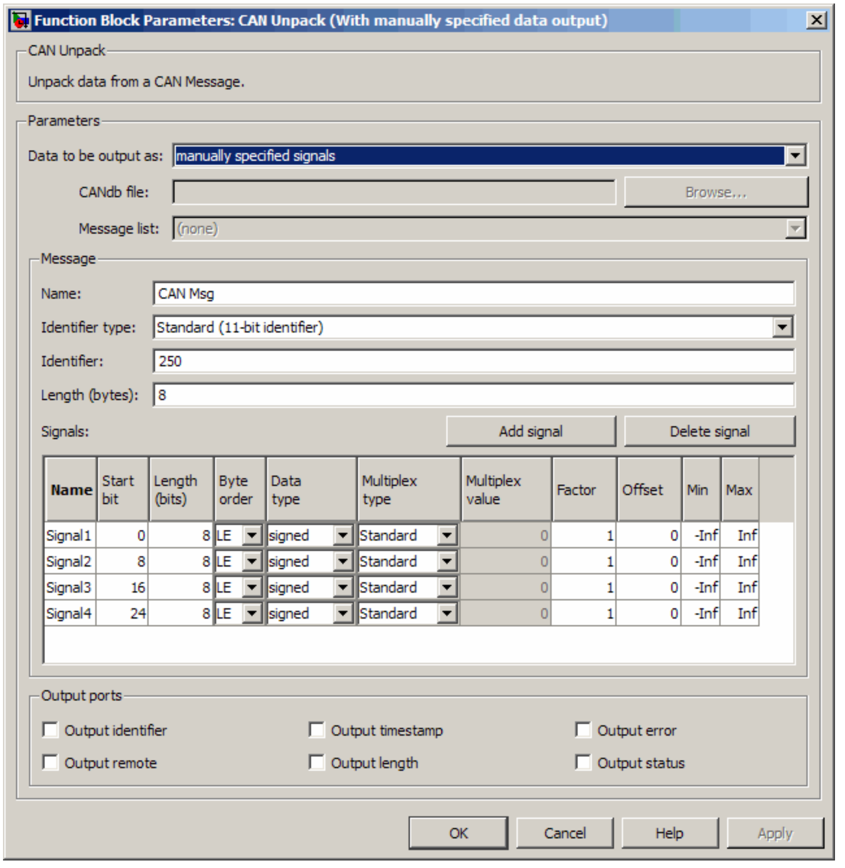


Parameters

Data to be output as

Select your data signal:

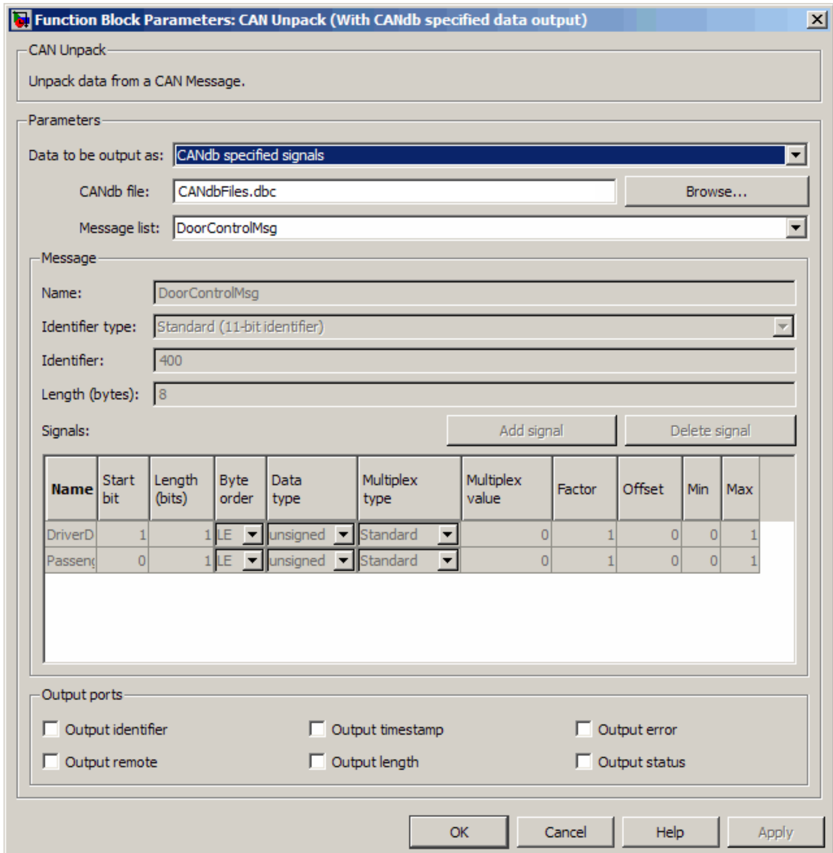
- **raw data**: Output data as a uint8 vector array. If you select this option, you only specify the message fields. All other signal parameter fields are unavailable. This option opens only one output port on your block.
- **manually specified signals**: Allows you to specify data signals. If you select this option, use the `Signals` table to create your signals message manually.



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.

Note You can specify a CAN database file only on a Windows 32–bit machine.



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 appropriate 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 any 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 all 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 any 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.

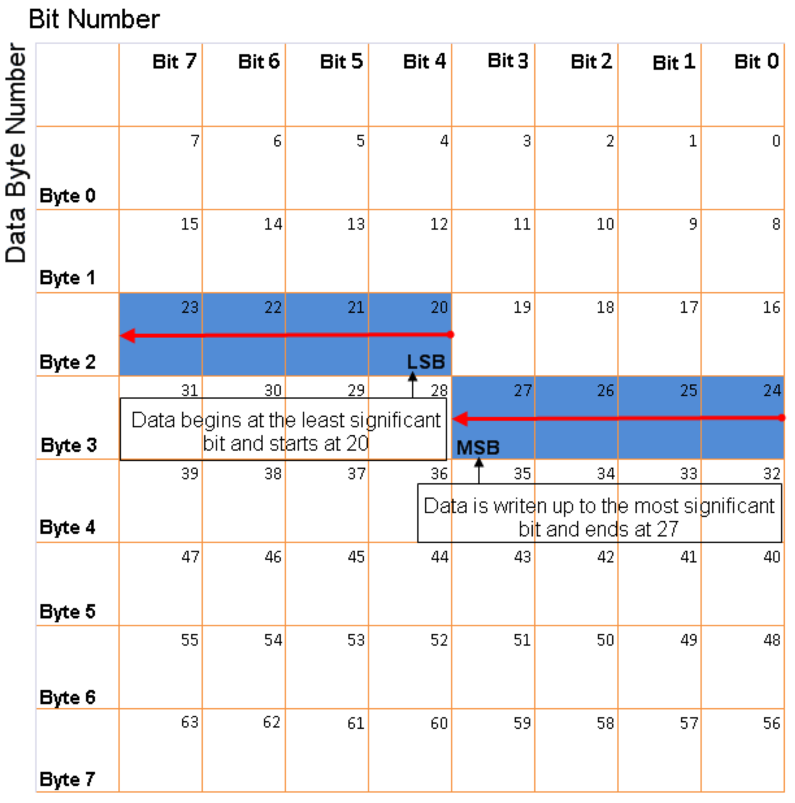
Length (bits)

Specify the number of bits the signal occupies in the message. The length must be an integer from 1 through 64.

Byte order

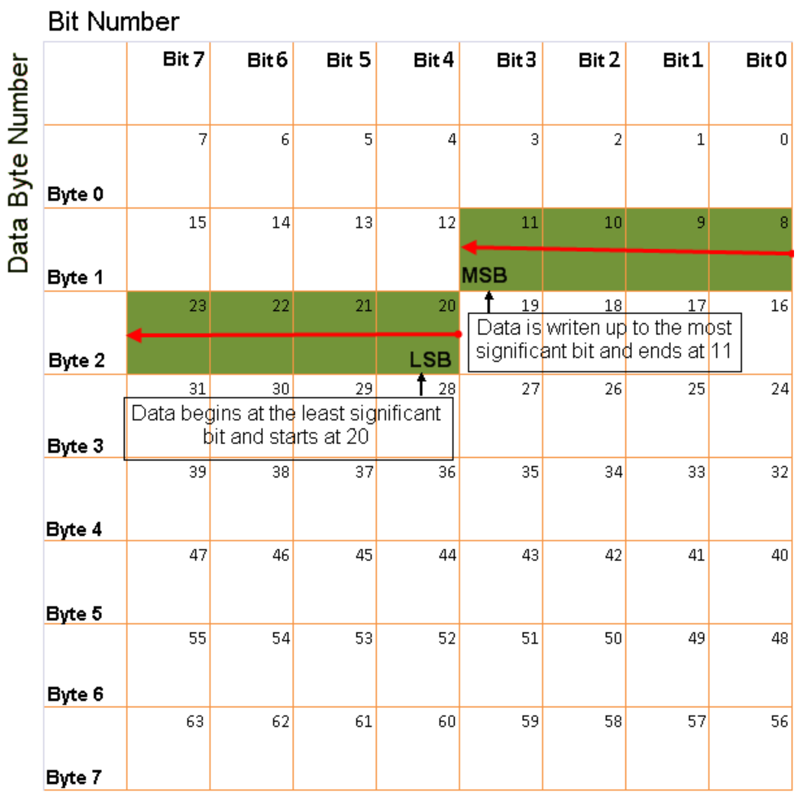
Select either of the following options:

- 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.



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.



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 always unpacked at each timestep.
- `Multiplexor`: The `Multiplexor` signal, or the mode signal is always 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 5-652 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 5-652 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 any number for the minimum value. See "Conversion Formula" on page 5-652 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 any number for the maximum value. See "Conversion Formula" on page 5-652 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 any **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

```
physical_value = raw_value *   Factor + 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

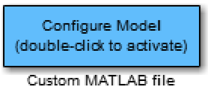
Purpose

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 "Creating 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

Purpose

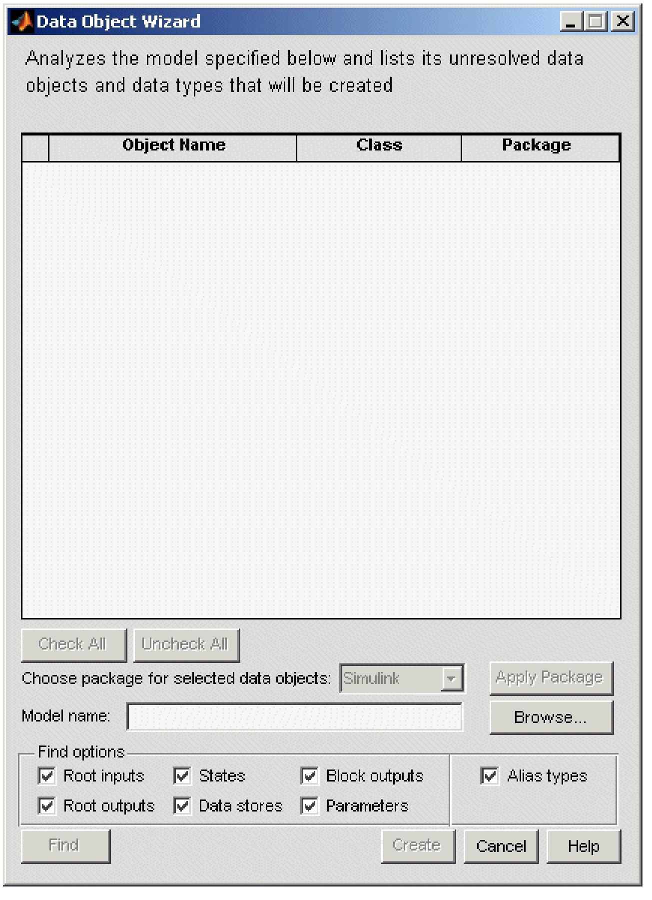
Simulink data object wizard for creating potential Simulink data objects

Library

Module Packaging

Description

When you add a Data Object Wizard block to your Simulink model and double-click it, the Data Object Wizard is launched:



The Data Object Wizard allows you to determine quickly which model data is not associated with Simulink data objects and to create and associate data objects with the data.

For detailed information about using the Data Object Wizard, see “Data Object Wizard” in the Simulink documentation and “Creating Simulink Data Objects with Data Object Wizard” in the Embedded Coder documentation.

You can also launch the Data Object Wizard by entering `dataobjectwizard` at the MATLAB command line or by selecting **Data Object Wizard** from the **Tools** menu of your model.

Example

For an example of a model that incorporates the Data Object Wizard block, see `rtwdemo_mpf`.

See Also

“Data Object Wizard” in the Simulink documentation

“Creating Simulink Data Objects with Data Object Wizard” in the Embedded Coder documentation

in the Embedded Coder documentation

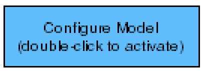
Purpose

Automatically update active configuration parameters of parent model for ERT fixed-point code generation

Library

Configuration Wizards

Description



ERT (optimized for fixed-point)

When you add an ERT (optimized for fixed-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `ERT (optimized for fixed-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom 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

Purpose

Automatically update active configuration parameters of parent model for ERT floating-point code generation

Library

Configuration Wizards

Description

When you add an ERT (optimized for floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for floating-point code generation with the ERT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `ERT (optimized for floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom 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

Purpose

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled

Library

Configuration Wizards

Description

When you add a GRT (debug for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed/floating-point code generation, with TLC debugging options enabled, with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `GRT (debug for fixed/floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom 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

Purpose

Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

Library

Configuration Wizards

Description

When you add a GRT (optimized for fixed/floating-point) block to your Simulink model and double-click it, a predefined MATLAB script executes and automatically configures the model parameters optimally for fixed/floating-point code generation with the GRT target. You can also set a block option to invoke the build process after configuring the model.

After double-clicking the block, you can verify that the model parameter values have changed by opening the Configuration Parameters dialog box and examining the settings.

Note You can include more than one Configuration Wizard block in your model. This provides a quick way to switch between configurations.

Parameters

Configure the model for

Value selected from

- `ERT (optimized for fixed-point)`
- `ERT (optimized for floating-point)`
- `GRT (optimized for fixed/floating-point)`
- `GRT (debug for fixed/floating-point)`
- `Custom`

For this block, `GRT (optimized for fixed/floating-point)` is selected by default.

Configuration function

Grayed out unless **Configure the model for** is set to `Custom`. This parameter is used with the Custom 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)

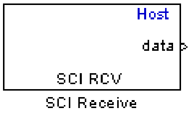
"Wizard" in the Embedded Coder documentation

Purpose

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.

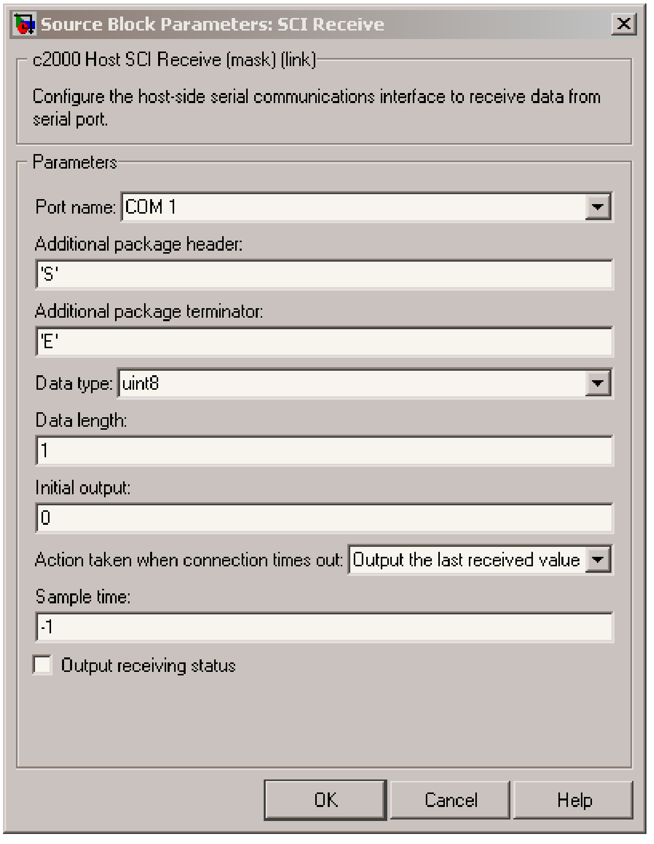
Acceptable data types are single, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The number of bytes in each data type is listed in the following table:

Data Type	Byte Count
`single`	4 bytes
`int8` and `uint8`	1 byte
`int16` and `uint16`	2 bytes
`int32` and`uint32`	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 may use any 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 Any additional packager header or terminator must match the additional package header or terminator 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 may use any 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.

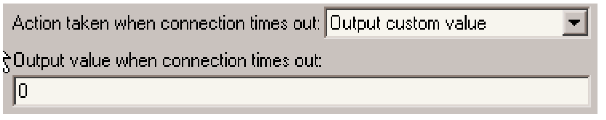
Data length

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.

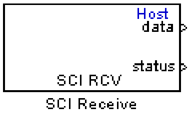
Action Taken when connection times out
Specify what to output if a connection time-out occurs. If "Output the last received value" is selected, the last received value is what is output, unless none has yet been received , in which case the **Initial output** is considered the last received value.

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 "Asynchronous Interrupt Processing" for a discussion of block placement and other necessary settings.

Output receiving status
When this field is checked, the SCI Receive block adds another output port for the transaction status, and appears as shown in the following figure.



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

"SCI_A, SCI_B, SCI_C" on page 5-900

Host SCI Setup

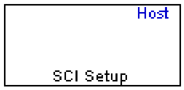
Purpose

Configure COM ports for host-side SCI Transmit and Receive blocks

Library

Embedded Coder/ Embedded Targets/ Host Communication

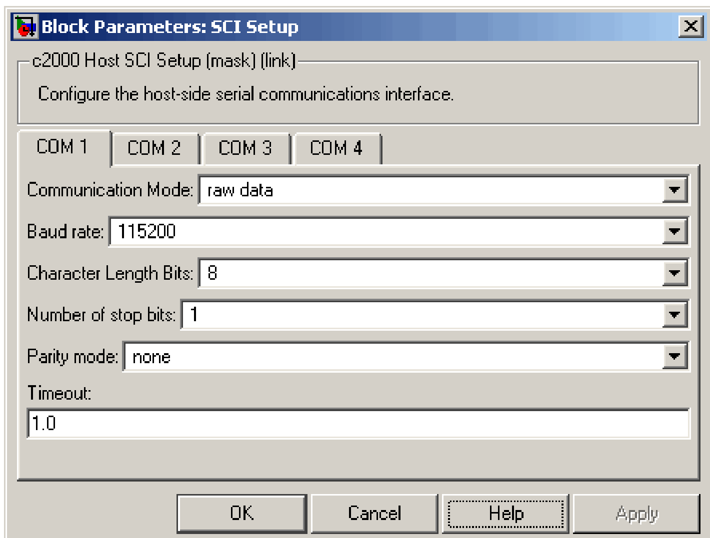
Description



SCI Setup

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. No deadlock condition can occur because there is no wait state. 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 any 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

- Ensures that data is received correctly (checksum)
- Ensures that data is actually received by target
- Ensures 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 any value 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 actually suspends processing for the length of the time-out, and you will not be able to perform any Simulink action. If the time-out is set for a long period of time, it may appear that Simulink has frozen.

See Also

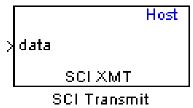
"SCI_A, SCI_B, SCI_C" on page 5-900

Purpose

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.

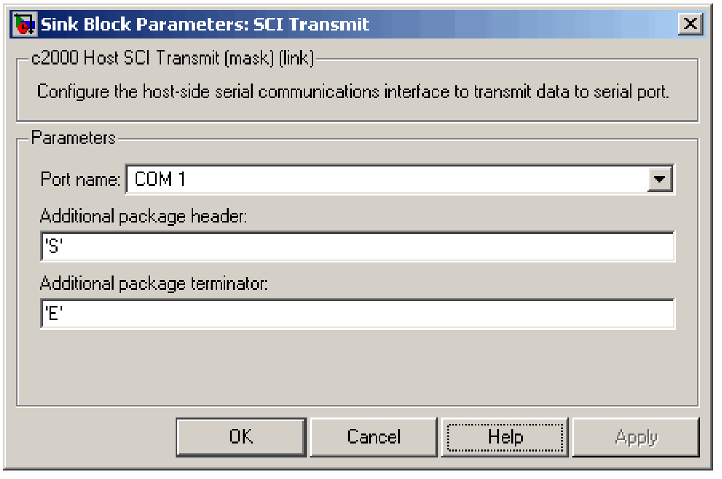
Acceptable data types are `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The byte size of each data type is as follows:

Data Type	Byte Count
`single`	4 bytes
`int8` & `uint8`	1 byte
`int16` & `uint16`	2 bytes
`int32` & `uint32`	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 may use any 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 Any additional packager header or terminator must match the additional package header or terminator 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 may use any 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

"SCI_A, SCI_B, SCI_C" on page 5-900

Purpose

Create free-running task

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices SHARC/ Scheduling

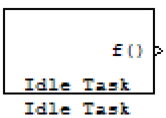
Embedded Coder/ Embedded Targets/ Processors/ Analog Devices TigerSHARC/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ Scheduling

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Scheduling

Description



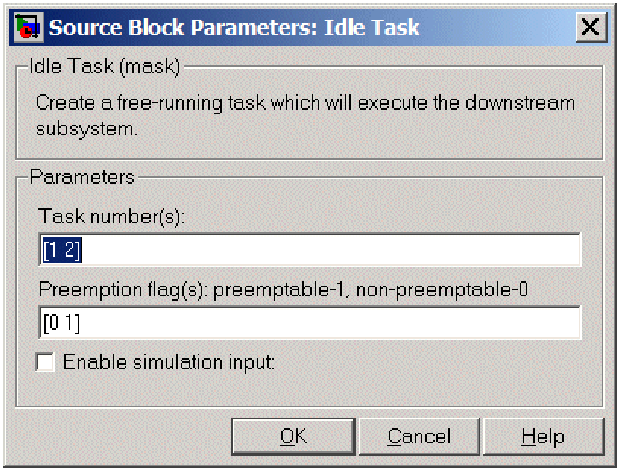
The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. All tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

Vectorized Output

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. Any 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 all 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 no more than 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 all 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 all 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.

Purpose

Configure AUTOSAR client port to access Basic Software or application software components

Library

Embedded Coder/ AUTOSAR

Description

Use this block to configure an AUTOSAR client port for your Simulink model, which provides access to Basic Software or application software components:

1. Copy or drag this block from the AUTOSAR library into your model.
2. Double-click the block to open the Invoke AUTOSAR Server Operation dialog box.
3. Specify the parameters and click **OK**. This action updates the number of inports and outports to match the operation prototype.
4. Connect this block to other blocks in your model as required.
5. Save and build the model to generate AUTOSAR-compliant code and XML files.

Note If you run a SIL simulation with a model that contains an Invoke AUTOSAR Server block, the software sets the return arguments from the block to ground values.

Simulink does not support pointer data types. If you want to pass a `NULL` pointer as an input argument to your operation:

1. Specify the data type of the argument as `uint8`.
2. Connect a constant signal with data type `uint8` and value 0 to the corresponding input port of the block.
3. Check that your client-server interface XML file specifies the argument as an array with data type `uint8`.

Parameters

Client port name

Must be a valid AUTOSAR short-name identifier.

Operation prototype

Controls the type and number of inports and outports of the block, and must be of the form:

```
operation(prt1 datatype1 arg1, prt2 datatype2 arg2, ...
prtN datatypeN argN, ...)
```

- `operation` — Name of the operation
- `prtN`. Either `IN` or `OUT`, which indicates whether data passes into or out of the function.
- `datatypeN` — A string indicating data type, which can be an AUTOSAR basic data type or record, Simulink data type, or array.
- `argN` — Name of the argument

Interface path

The reference path for the client-server interface XML file that you provide.

Server type

You select the value from:

- `Application software` — For communication with an application software component.
- `Basic software` — For communication with AUTOSAR Basic Software.

For this block, `Application software` is the default.

Show error status

If you select this, client port receives error status of client-server communication.

Sample time (-1 for inherited)

To inherit the sample time, set this parameter to -1.

See Also

Mode Switch for Invoke AUTOSAR Server Operation

"Configuring Client-Server Communication", `rtwdemo_autosar_clientserver_script`, and `rtwdemo_autosar_PIM_script` in the Embedded Coder documentation

Linux Audio Capture

Purpose

Capture ALSA audio from sound card and output data

Library

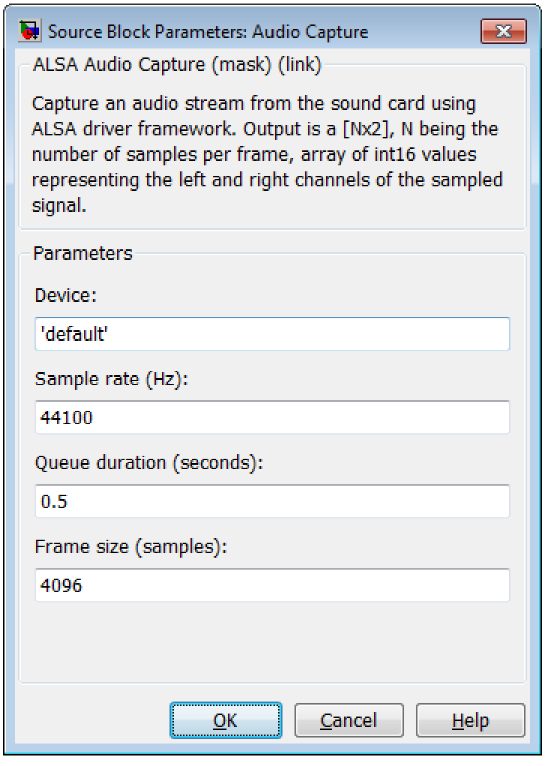
Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

Simulink Coder/ Desktop Targets/ Operating Systems/ Linux

Description

This block uses the ALSA driver framework to capture an audio stream from a sound card. It outputs the left and right channels of the signal as an [Nx2] frame of int16 values. N is the number of samples per frame.

Dialog



Device

Use the default ALSA device, or enter the name of a specific audio output device.

Entering `'default'` selects the ALSA device specified by an ALSA configuration file on your target Linux® system.

One of the following ALSA configuration files defines the default device:

- `/etc/asound.conf`, which defines system-wide options for all users
- `~/.asoundrc`, which overrides `/etc/asound.conf` for the current user

The entry that specifies the default device looks similar to this example:

```
pcm.!default {
    type hw
    card 0
    device 2
}
```

To enter the name of an alternate audio input device, review the `/proc/asound/cards` file on your target Linux system. For example, if `/proc/asound/cards` contained the following entries, you could set the value of **Device** to `'AudioPCI'` :

```
$ cat /proc/asound/cards

0 [Dummy          ]: Dummy - Dummy
                     Dummy 1

1 [VirMIDI        ]: VirMIDI - VirMIDI
                     Virtual MIDI Card 1
```

```
2 [AudioPCI   ]: ENS1371 - Ensoniq AudioPCI
                 Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

The default value for **Device** is `'default'`.

Sample rate (Hz)

Enter a value that matches the sample rate of the ALSA audio output.

By default, the sample rate of the ALSA output equals the output of the audio capture device. In this case, enter the sample rate of the audio capture device.

The `/etc/asound.conf` and `~/.asoundrc` files can configure ALSA to downsample the signal from the audio capture device. In this case, enter the downsample rate specified by the configuration files. For example, if one of the configuration files contained the following entry, you would set the value of **Sample rate (Hz)** to `16000` :

```
pcm_slave.sl3 {
    pcm ens1371
    format S16_LE
    channels 1
    rate 16000
}
pcm.complex_convert {
    type plug
    slave sl3
}
```

The default value for **Sample rate (Hz)** is `44100` Hz (44.1 kHz). The maximum rate equals the sampling rate of the audio capture device.

Queue duration (seconds)

Set the duration of the queue in seconds. This queue provides a software-based frame buffer between the ALSA output and the

Linux Audio Capture block. The queue prevents dropped data caused by temporary mismatches in the rate of data arriving and leaving the queue. Higher values can handle more significant mismatches, but such values also increase signal latency and memory usage.

The default value for **Queue duration (seconds)** is `0.5` seconds.

Frame size (samples)

Set the number of samples per frame in the output this block sends to your model. The default value for this parameter is `4096` samples.

References

http://www.alsa-project.org

See Also

http://www.alsa-project.org

Linux Audio Playback

Linux Task

Linux Audio Playback

Purpose

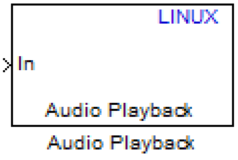
Send audio data stream to ALSA audio device output

Library

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux (linuxlib)

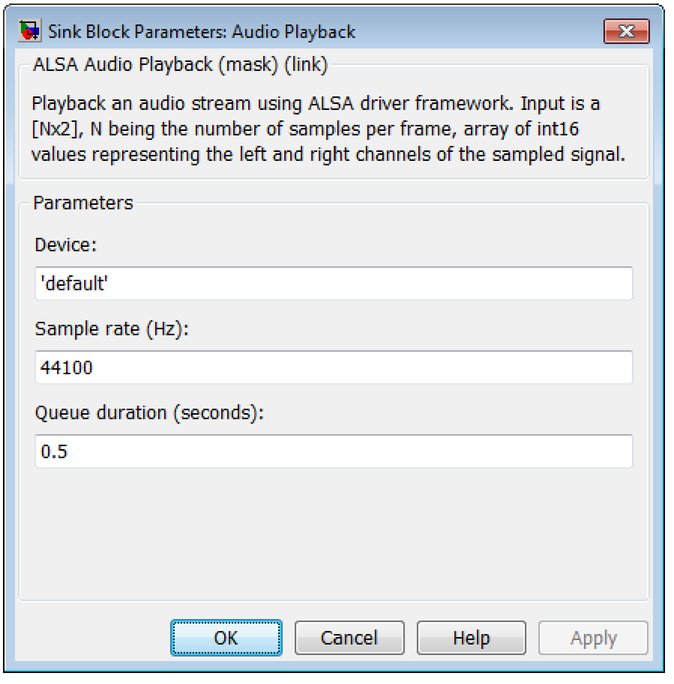
Simulink Coder/ Desktop Targets/ Operating Systems/ Linux

Description



This block takes a stream of audio data and sends it to the output jack of an ALSA audio device. The block input, **In**, takes the left and right channels of data as an [Nx2] frame of int16 values. N is the number of samples per frame.

Dialog



Device

Use the default ALSA device, or enter the name of a specific audio device.

Entering `'default'` selects the ALSA device specified by an ALSA configuration file on your target Linux system.

One of the following ALSA configuration files defines the default device:

- `/etc/asound.conf`, which defines system-wide options for all users
- `~/.asoundrc`, which overrides `/etc/asound.conf` for the current user

The entry that specifies the default device looks like this hypothetical example:

```
pcm.!default {
    type hw
    card 0
    device 2
}
```

To enter the name of an alternate audio device, consult the `/proc/asound/cards` file on your target Linux system. For example, if `/proc/asound/cards` contained the following hypothetical entries, you could set the value of **Device** to `'AudioPCI'`:

```
$ cat /proc/asound/cards

0 [Dummy       ]: Dummy - Dummy
                  Dummy 1

1 [VirMIDI     ]: VirMIDI - VirMIDI
                  Virtual MIDI Card 1
```

```
2 [AudioPCI   ]: ENS1371 - Ensoniq AudioPCI
                 Ensoniq AudioPCI ENS1371 at 0xe400, irq 11
```

The default value for **Device** is `'default'`.

Sample rate (Hz)

Enter a value that matches the sample rate of the ALSA audio output.

By default, the sample rate of the ALSA output is the same as the output of the audio capture device. In this case, enter the sample rate of the audio capture device.

The `/etc/asound.conf` and `~/.asoundrc` files can configure ALSA to downsample the signal from the audio capture device. In this case, enter the downsample rate specified by the configuration files. For example, if one of the configuration files contained the following hypothetical entry, you would set the value of **Sample rate (Hz)** to `16000` :

```
pcm_slave.sl3 {
    pcm ens1371
    format S16_LE
    channels 1
    rate 16000
}
pcm.complex_convert {
    type plug
    slave sl3
}
```

The default value for **Sample rate (Hz)** is `44100` Hz (44.1 kHz). The maximum rate is the sampling rate of the audio capture device.

Queue duration (seconds)

Set the duration of the queue in seconds. This queue provides a software-based frame buffer between the ALSA audio device and this block. The queue prevents dropped data caused by temporary mismatches in the rate of data arriving and leaving the queue. Higher values can handle more significant mismatches, but increase signal latency and memory usage.

The default value for **Queue duration (seconds)** is `0.5` seconds.

See Also

http://www.alsa-project.org

Linux Audio Capture

Linux Task

Linux Task

Purpose

Spawn task function as separate Linux thread

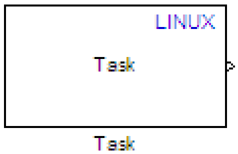
Library

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

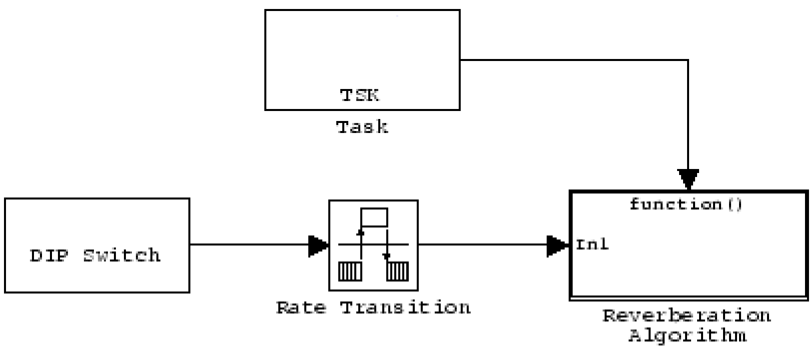
Simulink Coder/ Desktop Targets/ Operating Systems/ Linux

Description

This documentation will be updated.

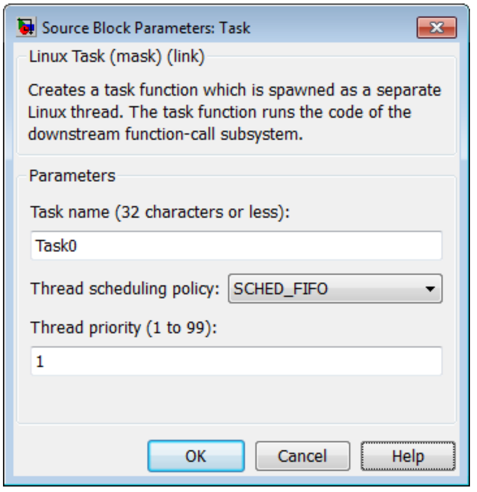


Use this block to create a task function that spawns as a separate Linux thread. The task function runs the code of the downstream function-call subsystem. For example:



Dialog

This documentation will be updated.



Task name

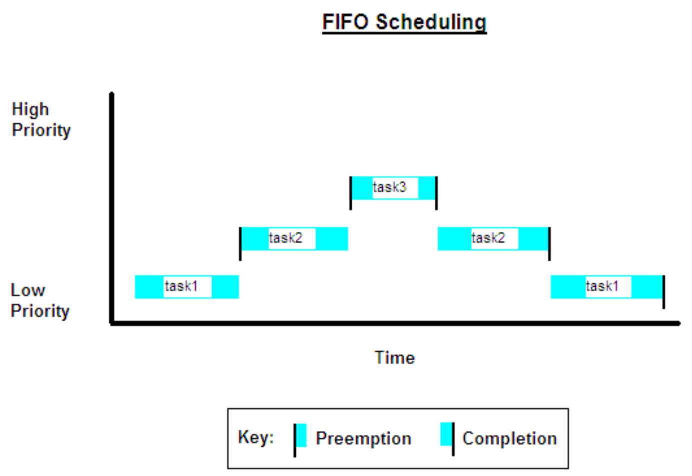
Assign a name to this task. You can enter up to 32 letters and numbers. Do not use standard C reserved characters, such as the / and : characters.

Thread scheduling policy

Select the scheduling policy that applies to this thread. You can choose from the following options:

- `SCHED_FIFO` enables a First In, First Out scheduling algorithm that executes real-time processes without time slicing. With FIFO scheduling, a higher-priority process preempts a lower-priority process. The lower-priority process remains at the top of the list for its priority and resumes execution when the scheduler blocks all higher-priority processes.

  For example, in the following image, task2 preempts task1. Then task3 preempts task2. When task3 completes, task2 resumes. When task2 completes, task1 resumes.



Selecting SCHED_FIFO, displays the **Thread priority** parameter, which you can set to a value from 1 to 99.

- SCHED_OTHER enables the default Linux time-sharing scheduling algorithm. You can use this scheduling for all processes except those requiring special static priority real-time mechanisms. With this algorithm, the scheduler chooses processes based on their dynamic priority within the static priority 0 list. Each time the process is ready to run and the scheduler denies it, the operating system increases that process's dynamic priority. Such prioritization ensures the scheduler serves the SCHED_OTHER processes in the correct order.

  Selecting SCHED_OTHER, hides the **Thread priority** parameter, and sets the thread priority to 0.

Thread priority (1 to 99)

When you set **Thread scheduling policy** to SCHED_FIFO, you can set the priority of the thread from 1 to 99 (low-to-high).

Higher-priority tasks can preempt lower-priority tasks.

See Also

Linux Audio Capture

Linux Audio Playback

Memory Allocate

Purpose

Allocate memory section

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices SHARC/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices TigerSHARC/ Memory Operations

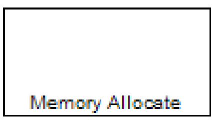
Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC55xx MPC74xx/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Memory Operations

Description



Memory Allocate

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.

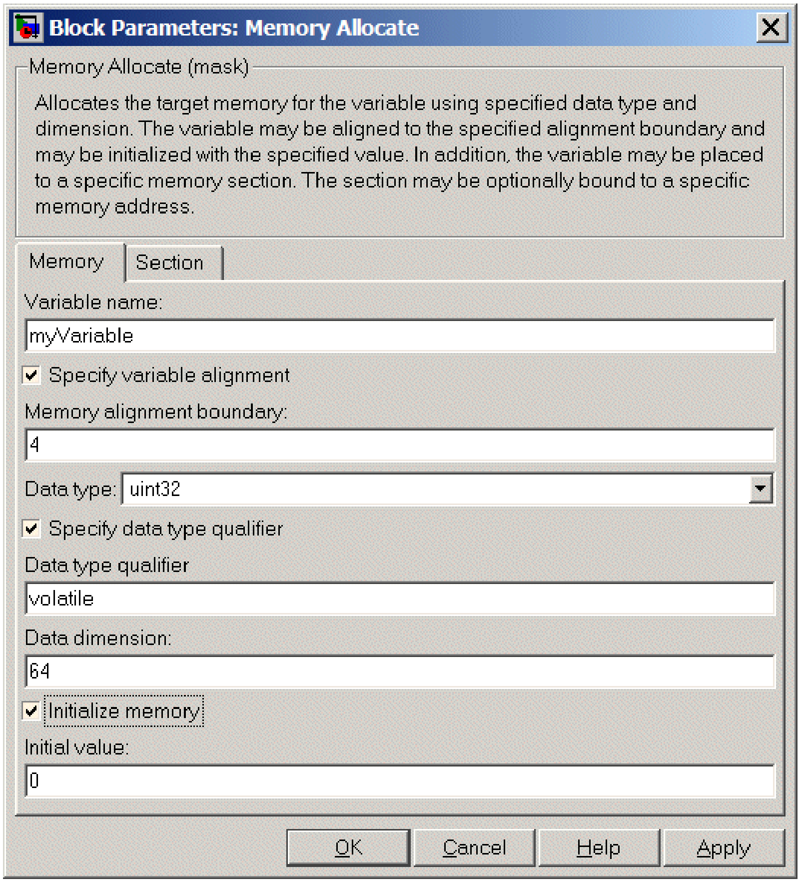
Note When using this block with Green Hills MULTI IDE and Blackfin® processors, set the `-no_discard_zero_initializers` option.

Dialog Box

The block dialog box comprises multiple tabs:

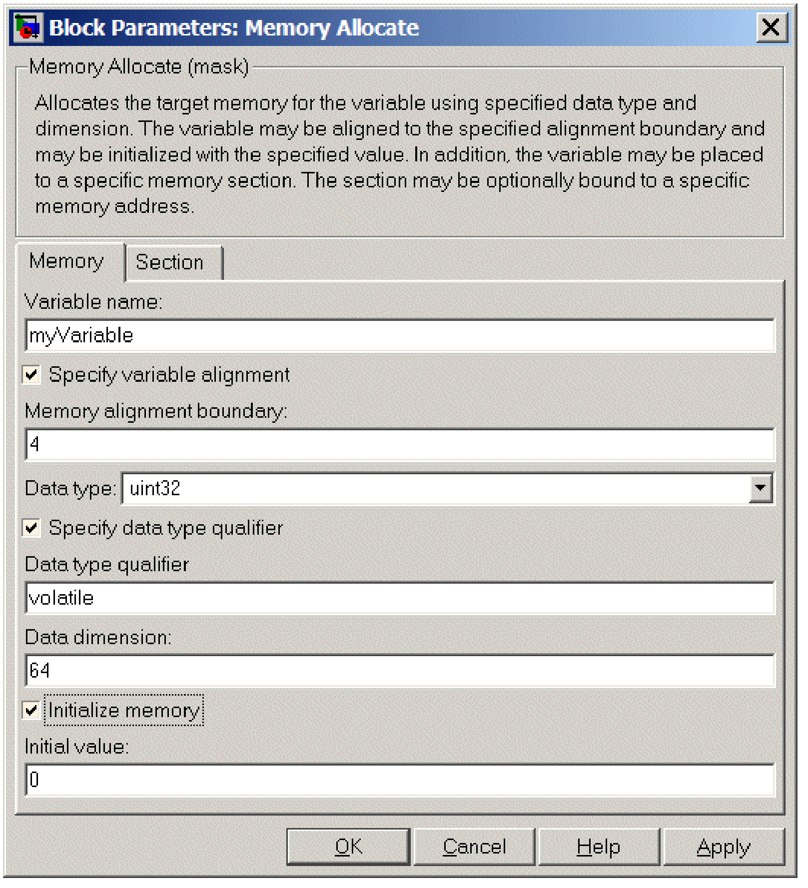
- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

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.



The following sections describe the contents of each pane in the dialog box.

Memory Parameters



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.

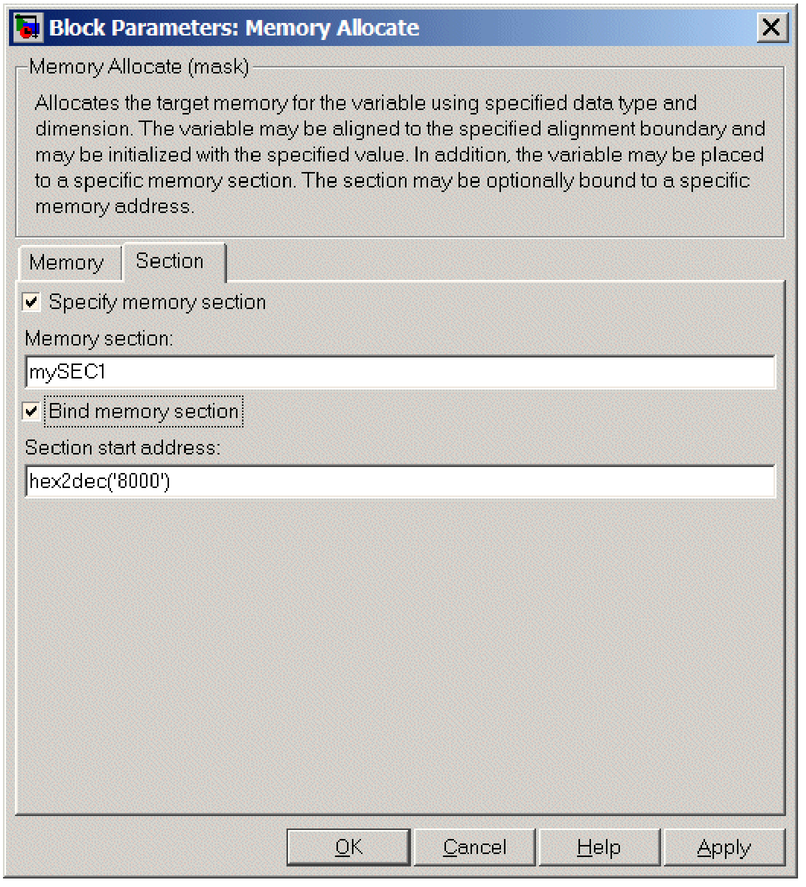
Data type qualifier
After you select **Specify data type qualifier**, you enter the desired qualifier here. `Volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

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 sufficient 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

Purpose

Copy to and from memory section

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices Blackfin/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices SHARC/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices TigerSHARC/ Memory Operations

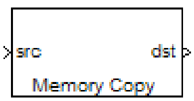
Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC55xx MPC74xx/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C2000/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C5000/ Memory Operations

Embedded Coder/ Embedded Targets/ Processors/ Texas Instruments C6000/ Memory Operations

Description



Memory Copy

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 all 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 any operation. The block output is not defined.

Copy Memory

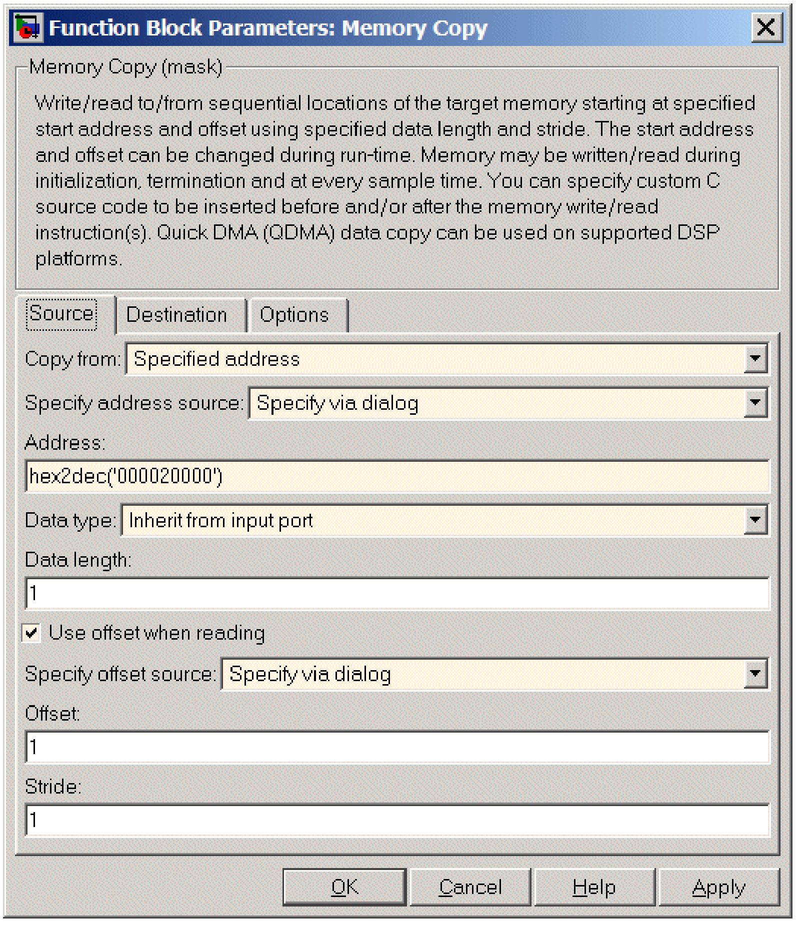
When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

Dialog Box

The block dialog box contains multiple tabs:

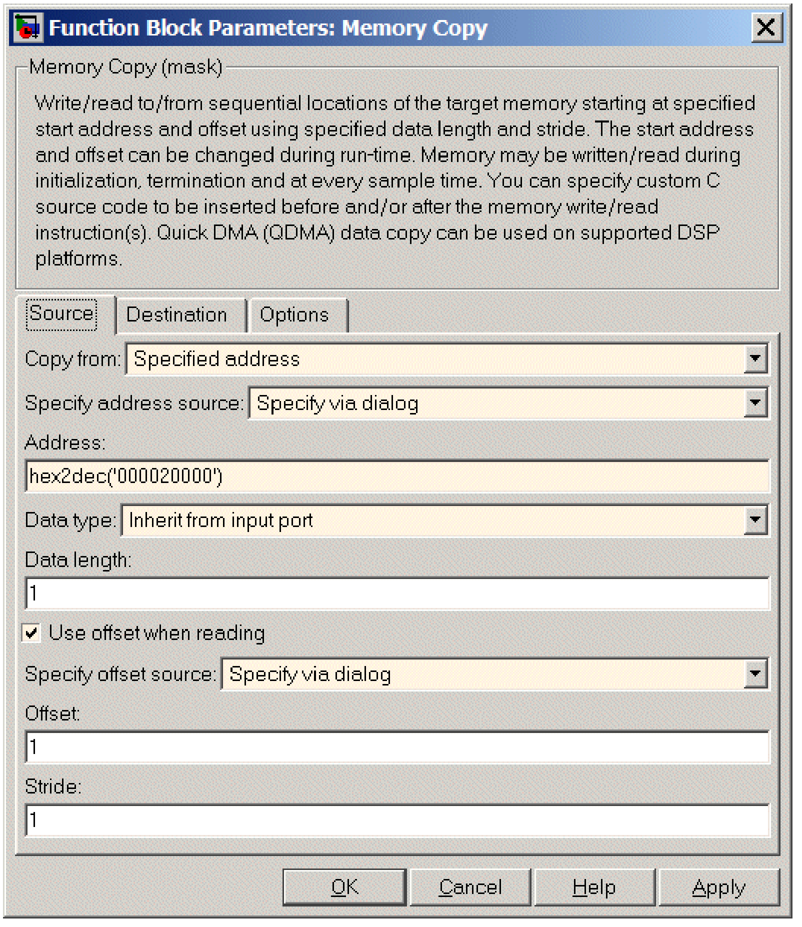
- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various 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



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 proper 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.

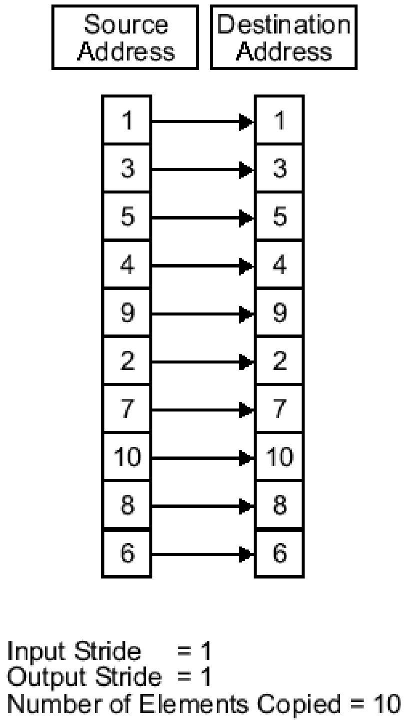
Offset

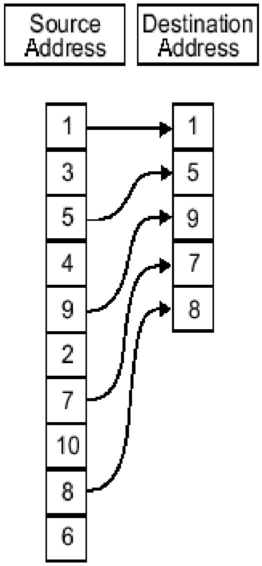
Offset tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the 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.

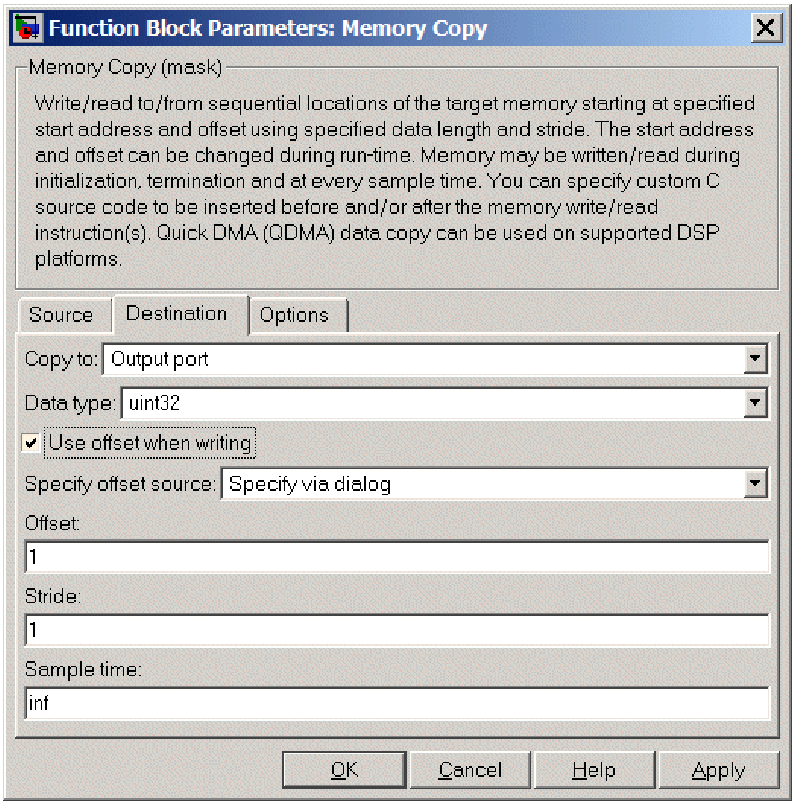
The next two figures help explain the stride concept. In the first figure you see data copied without any stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.


Source Address
Destination Address
1
3
5
4
9
2
7
10
8
6
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 proper 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.

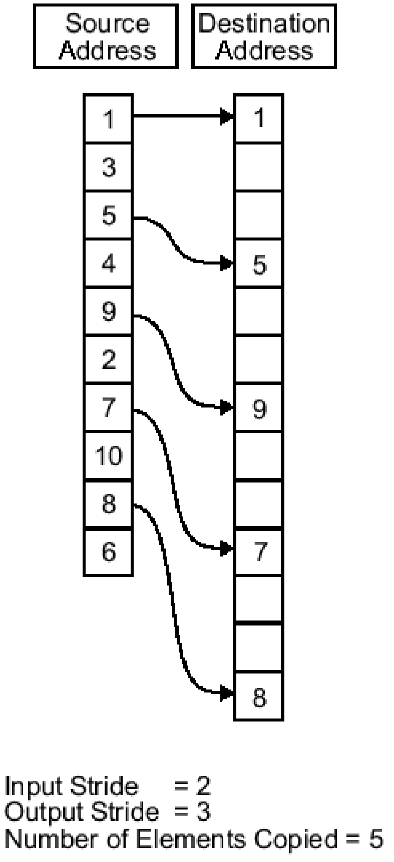
Data type
: Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `inherit from source` for inheriting the data type for the variable from the block input port.

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.

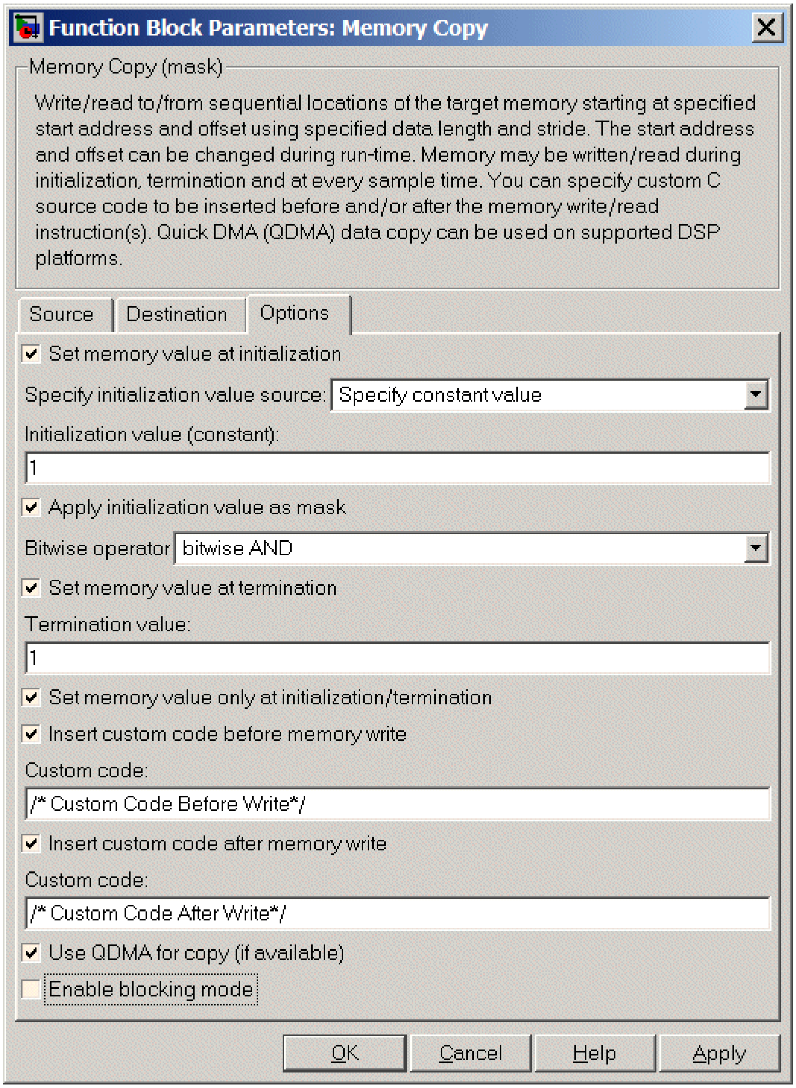
This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.



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, if there is one, or the Simulink software model (when there are no input ports on the block). Enter the sample time in seconds as you need.

Options Parameters



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. Enter any value that meets your needs.
- `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. Any real value that meets your needs is acceptable.

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. Any symbol that meets your needs and is in the symbol table for the program is acceptable. 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 any 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

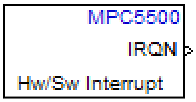
MPC5500 Interrupt

Purpose

Generate Interrupt Service Routine

Library

Block Library: idelinklib_ghsmulti

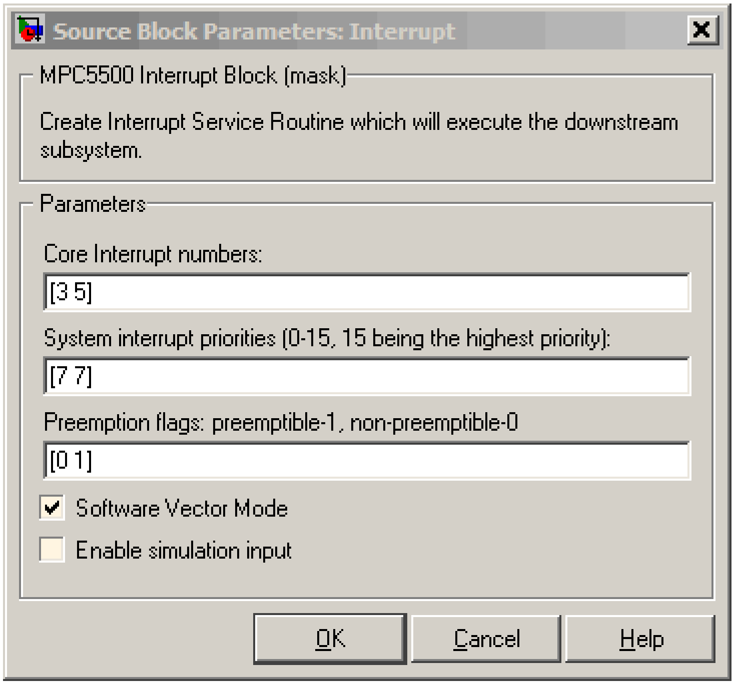


Interrupt

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. Core interrupts trigger the ISRs. System interrupts trigger the core interrupts.

Dialog Box



Core interrupt numbers

Specify a vector of interrupt numbers for the interrupts to install. The block services these interrupts. When your model or code raises one of these interrupts, either through hardware or software, this block reacts to the interrupt and runs the associated downstream block or function. The valid range or interrupts depends on the processor. For example, MPC5553 processors support 212 interrupts. MPC5554 processors support 308 interrupts. Each interrupt in the row vector must be unique. Interrupts that you do not specify in this parameter cause system failures if your project raises them.

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.

System interrupt priorities (0–15, 15 being the highest priority)

Each output of the HW/SW Interrupt block drives a downstream block (for example, a function call subsystem). Simulink task priority specifies the Simulink priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Core interrupt numbers**. In the default settings shown in the figure, interrupts 3 and 5 have the same priority value—7.

Proper code generation requires rate transition code (see Rate Transitions and Asynchronous Blocks). The task priority values make certain there is 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.

If multiple interrupts share the same priority and are asserted simultaneously, the block selects the lowest numbered interrupt first.

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.

You cannot set a task that has priority higher than the base rate to be preemptable.

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 `5` in **Interrupt numbers** is not preemptible and the priority `8` interrupt can be preempted.

Software vector mode

Select this option to put the block and processor in software vector mode. Enabling this option creates a common interrupt handler. Clearing this option puts the processor in hardware vector mode. Refer to the MULTI documentation for more information about the modes.

Enable simulation input

When you select this option, Simulink adds an input port to the HW/SW Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

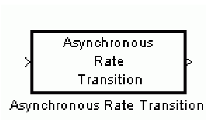
Purpose

Transfer data between timer-based task and asynchronous task, ensuring data integrity

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Interrupts

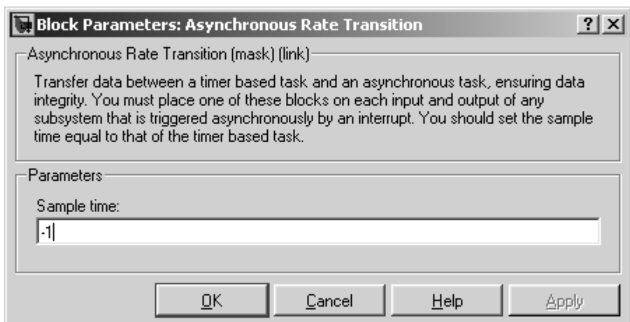
Description



The Asynchronous Rate Transition block is used when reading or writing signals attached to an asynchronous subsystem. An asynchronous subsystem is one which is driven by an interrupt function call trigger. The subsystem is run in the context of an interrupt and not in the context of the model. You must place one of these blocks on each input and output of any subsystem that is triggered asynchronously by an interrupt.

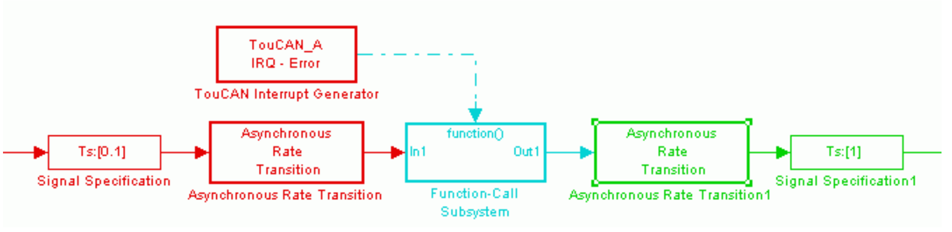
The Asynchronous Rate Transition block copies the signal from input to output while disabling interrupts. In doing this, blocks outside the subsystem that want access to the signal do not get interrupted while reading or writing a signal and end up with corrupt data.

Dialog Box



Sample time

You should set the sample time equal to that of the timer based task, as shown in the following example model.



See also the MPC5xx TouCAN Interrupt Generator.

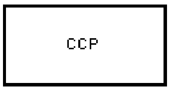
Purpose

Implement CAN Calibration Protocol (CCP) standard

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

Description



CAN Calibration Protocol

The CAN Calibration Protocol (MPC555) block provides an implementation of a subset of the CAN Calibration Protocol (CCP) Version 2.1. CCP is a protocol for communicating between the target processor and the host machine over CAN. In particular, a calibration tool (see “Compatibility with Calibration Packages” on page 5-732) running on the host can communicate with the target, allowing remote signal monitoring and parameter tuning.

This block processes Command Receive Object (CRO) messages and outputs the resulting Data Transmission Object (DTO) and Data Acquisition (DAQ) messages.

For more information on CCP, refer to *ASAM Standards: ASAM MCD: MCD 1a* on the Association for Standardization of Automation and Measuring Systems (ASAM) Web site at `http://www.asam.de`.

You can see an example illustrating how to use the CAN Calibration Protocol (MPC555) block in the `mpc555rt_ccp` demo.

Note this block is entirely CAN triggered, and so is only designed for the Real-Time Target (CAN is disabled during PIL and SIL simulation.)

Using the DAQ Output

Note The CCP Data Acquisition (DAQ) List mode of operation is only supported with the Embedded Coder product. If this is not available then custom storage classes `canlib.signal` are ignored during code generation: this means that the CCP DAQ Lists mode of operation cannot be used.

You can use the CCP Polling mode of operation with or without Embedded Coder software.

The DAQ output is the output for any CCP DAQ lists that have been set up. You can use the ASAP2 file generation feature of the RT target to

- Set up signals to be transmitted using CCP DAQ lists.
- Assign signals in your model to a CCP event channel automatically. See "Generating an ASAP2 File"

Once these signals are set up, event channels then periodically fire events that trigger the transmission of DAQ data to the host. When this occurs, CAN messages with the appropriate CCP/DAQ data appear on the DAQ output, along with an associated function call trigger.

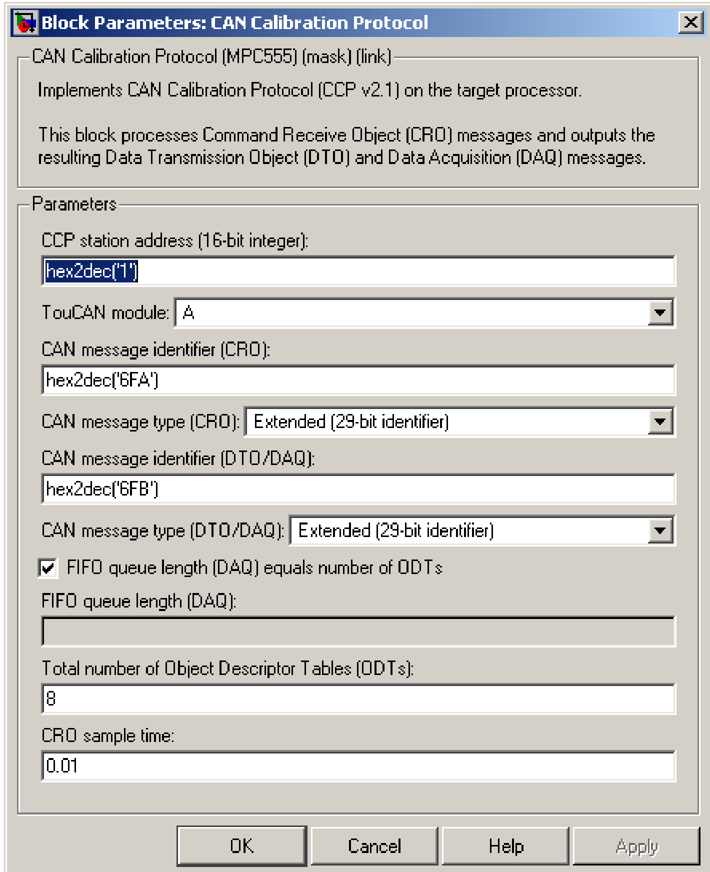
It is the responsibility of the calibration tool (see "Compatibility with Calibration Packages" on page 5-732) to use CCP commands to assign an event channel and data to the available DAQ lists, and to interpret the synchronous response.

Using DAQ lists for signal monitoring has the following advantages over the polling method:

- There is no need for the host to poll for the data. Network traffic is halved.
- The data is transmitted at the correct update rate for the signal. Therefore there is no unnecessary network traffic generated.
- Data is consistent. The transmission takes place after the signals have been updated, so there is no risk of interruptions while sampling the signal.

Note The Embedded Coder product does not currently support event channel prescalers.

Dialog Box



CAN station address (16 bit integer)

The station address of the target. The station address is interpreted as a `uint16`. It is used to distinguish between different targets. By assigning unique station addresses to targets sharing the same CAN bus, it is possible for a single host to communicate with multiple targets.

TouCAN module

Choose A or B.

CAN message identifier (CRO)
Specify the CAN message identifier for the incoming Command Receive Object (CRO) message you want to process.

CAN message type (CRO)
The incoming message type. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

CAN message identifier (DTO/DAQ)
The message identifier is the CAN message ID used for Data Transmission Object (DTO) and Data Acquisition (DAQ) message outputs. It is also used for transmitting messages to the host during the software-induced CAN download (soft boot). See "Extended Functionality" on page 5-733.

CAN message type (DTO/DAQ)
The message type to be transmitted by the DTO and DAQ outputs. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

FIFO queue length (DAQ) equals number of ODTs
Leave this check box selected to automatically set the FIFO queue length equal to the number of Object Descriptor Tables (ODTs) (recommended). Clear the check box to set the length of the FIFO queue manually.

FIFO queue length (DAQ)
Specify the FIFO queue length manually. This is enabled if you clear the check box to set the queue length automatically.

Total number of Object Descriptor Tables (ODTs)
The default number of Object Descriptor Tables (ODTs) is 8. These ODTs are shared equally between all available DAQ lists. You can choose a value between 0 and 254, depending on how many signals you want to log simultaneously. You must make sure you allocate at least 1 ODT per DAQ list, or your build will fail. The calibration tool will give an error message if there are too few ODTs for the number of signals you specify for monitoring. Be aware that too many ODTs can make the sample time overrun.

If you choose more than the maximum number of ODTs (254), the build will fail.

A single ODT uses 56 bytes of memory. Using all 254 ODTs would require over 14 KB of memory, a large proportion of the available memory on the target. To conserve memory on the target the default number is low, allowing DAQ list signal monitoring with reduced memory overhead and processing power.

As an example, if you have five different rates in a model, and you are using three rates for DAQ, then this will create three DAQ lists and you must make sure you have at least three ODTs. ODTs are shared equally among DAQ lists, and therefore you will end up with one ODT per DAQ list. With less than three ODTs you get zero ODTs per DAQ list and the behavior is undefined.

Taking this example further, say you have three DAQ lists with one ODT each, and start trying to monitor signals in a calibration tool. If you try to assign too many signals to a particular DAQ list (that is, signals requiring more space than seven bytes (one ODT) in this case), then the calibration tool will report this as an error.

For more information on DAQ lists, see "Data Acquisition (DAQ) List Configuration".

CRO sample time
Sample time at which to check for incoming Command Receive Object (CRO) messages.

Supported CCP Commands

The following CCP commands are supported by the CAN Calibration Protocol (MPC555) block:

- `CONNECT`
- `DISCONNECT`
- `DNLOAD`

- DNLOAD_6
- EXCHANGE_ID
- GET_CCP_VERSION
- GET_DAQ_SIZE
- GET_S_STATUS
- SET_DAQ_PTR
- SET_MTA
- SET_S_STATUS
- SHORT_UP
- START_STOP
- START_STOP_ALL
- TEST
- UPLOAD
- WRITE_DAQ

Compatibility with Calibration Packages

The above commands support

- Synchronous signal monitoring via calibration packages that use DAQ lists
- Asynchronous signal monitoring via calibration packages that poll the target
- Asynchronous parameter tuning via CCP memory programming

This CCP implementation has been tested successfully with the Vector-Informatik CANape calibration package running in both DAQ list and polling mode, and with the Accurate Technologies Inc. Vision calibration package running in DAQ list mode. (Note that Accurate

Technologies Inc. Vision does not support the polling mechanism for signal monitoring.)

Extended Functionality

The CAN Calibration Protocol (MPC555) block also supports the PROGRAM_PREPARE command. This command is an extension of CCP that allows the automatic download of new code into the target memory. This removes the requirement for a manual reset of the processor. On receipt of the PROGRAM_PREPARE command, the target will reboot and begin the CAN download process. This lets you download new application code to RAM or flash memory, or download new boot code to flash memory. See "Downloading Boot or Application Code via CAN Without Manual CPU Reset".

Note The CAN message identifiers of the CCP messages incoming to the target (Command Receive Object (CRO) messages) and the messages outgoing from the target (Data Transmission Object (DTO) or DAQ) are specified in the block mask for the CAN Calibration Protocol (MPC555) block. These message identifiers are used as the CAN identifiers for the download process after a PROGRAM_PREPARE reboot. The type of CAN message used for this PROGRAM_PREPARE download process is always Extended (29-bit identifier).

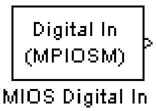
Purpose

Input driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Modular Input/Output System (MIOS1)

Description

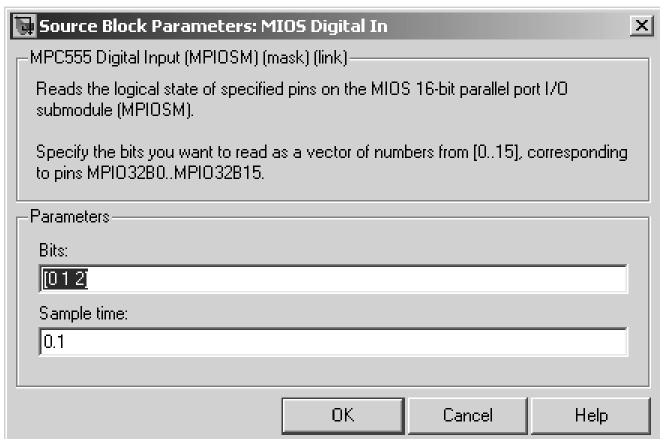


The MIOS Digital In block reads the state of selected pins (bits) on the MIOS 16-bit Parallel Port I/O Submodule (MPIOSM) of the MPC555. The **Bits** field specifies a vector of numbers in the range `0..15`, corresponding to pins `MPIO32B0..MPIO32B15` on the MPIOSM.

The output of the block is a wide vector representing the logic state of the pins referenced in the **Bits** field. When the signal on a given pin is a logical 1, the block output element will be equal to 1; otherwise the block output element will equal zero.

Refer to section 15.13, "MIOS 16-bit Parallel Port I/O Sub module (MPIOSM)," in the *MPC555 User's Manual* for further information.

Dialog Box



Bits

A vector of numbers in the range `0..15`. Each number corresponds to a pin (`MPIO32B0..MPIO32B15`) on the MPIOSM.

Sample time

Sample time of the block.

MPC5xx MIOS Digital Out

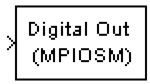
Purpose

Output driver for MIOS 16-bit Parallel Port I/O Submodule (MPIOSM)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Modular Input/Output System (MIOS1)

Description



MIOS Digital Out

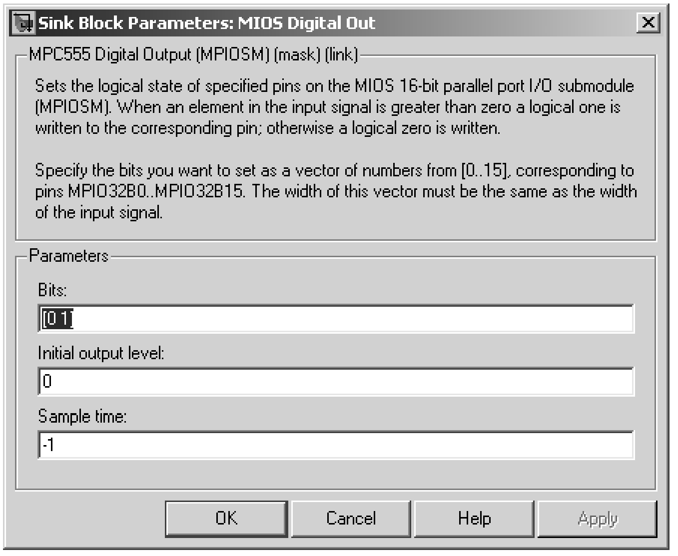
The MIOS Digital Out block sets the state of selected pins (bits) on the MIOS 16-bit Parallel Port I/O Submodule (MPIOSM) of the MPC555. The **Bits** field specifies a vector of numbers in the range `0..15`, corresponding to pins `MPIO32B0..MPIO32B15` on the MPIOSM.

The input to the block is a wide vector with one signal element per pin. When the input signal is greater than zero, a logical 1 is written to the corresponding pin. When the input signal is less than or equal to zero, a logical zero is written to the corresponding pin.

If you want to write to several digital output pins at the same sample rate, using a single MIOS Digital Out block with a vector input signal will result in more efficient code. However, if you want to update different output pins at different sample rates, you must use a separate MIOS Digital Out block for each rate.

Refer to section 15.13, "MIOS 16-bit Parallel Port I/O Sub module (MPIOSM)," in the *MPC555 User's Manual* for further information.

Dialog Box



Bits

A vector of numbers in the range `0..15`. Each number corresponds to a pin (`MPIO32B0..MPIO32B15`) on the MPIOSM.

Initial output level

The value to be placed on the output pins at initialization. In setting this, the starting level is always known.

Sample time

The sample time of this block.

MPC5xx MIOS Digital Out (MPWMSM)

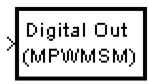
Purpose

Digital output via the MIOS Pulse Width Modulation Submodule (MPWMSM)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Modular Input/Output System (MIOS1)

Description

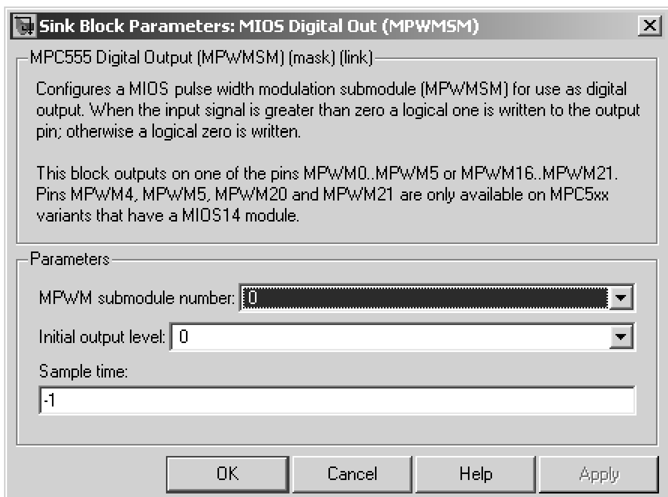


MIOS Digital Out (MPWMSM)

The MIOS Digital Out (MPWMSM) block is a device driver that lets you use the MIOS Pulse Width Modulation Submodule (MPWMSM) in *digital output mode*. In digital output mode, the Pulse Width Modulation (PWM) feature of the MPWMSM is turned off. When the input signal is greater than zero, a logical 1 is written to the output pin; otherwise a logical zero is written.

Refer to section 15.12, "MIOS Pulse Width Modulation Submodule (MPWMSM)," in the *MPC555 User's Manual* for further information on the parameters described below.

Dialog Box



MPWM submodule number
Select a PWM submodule for output. Note that modules 4, 5, 20 and 21 are for the MPC56x (561-6) only. If you select one of these modules and MPC555 is the processor selected in the Resource Configuration block, then an error will be thrown on updating the model.

Initial output level
The value to be placed on the output pins at initialization. In setting this, the starting level is always known.

Sample time
Sample time of the block.

Invert output polarity
Switches the output level for logic one and zero.

MPC5xx MIOS Pulse Width Modulation Out

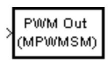
Purpose

Output driver for MIOS Pulse Width Modulation Submodule (MPWMSM)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Modular Input/Output System (MIOS1)

Description



MIOS Pulse Width Modulation Out

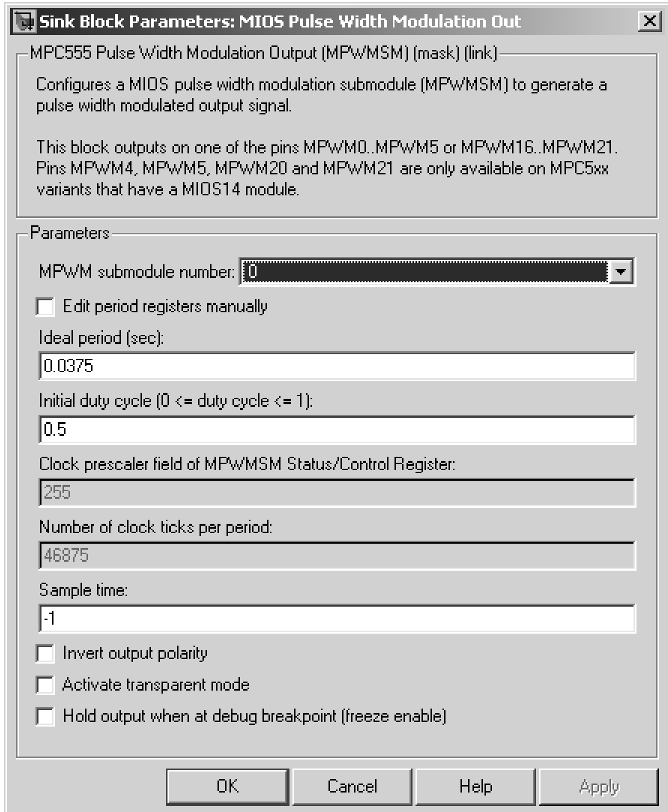
The MIOS Pulse Width Modulation Out block is used for Pulse Width Modulation (PWM) output from the MIOS Pulse Width Modulation Submodule (MPWMSM). A PWM signal is a rectangular waveform whose period is constant but whose duty cycle can be varied, under control of a modulator signal, between 0% and 100%.

The MIOS Pulse Width Modulation block input signal acts as the modulator, controlling the duty cycle of the signal on the output pin. The input signal is multiplied by the period register value, and saturates if outside 0-1. When the input signal value is 0, the output signal's duty cycle is 0%. When the input signal value is 1, the output signal's duty cycle is 100%.

There are two possible methods for calculating the period of the waveform. You can either control the scaling registers directly, or enter the desired (ideal) period and the mask will solve for the best values for the scaling registers.

Refer to section 15.12, "MIOS Pulse Width Modulation Submodule (MPWMSM)," in the *MPC555 User's Manual* for further information on the parameters described below.

Dialog Box



MPWM submodule number

Select a PWM submodule for output. Note that modules 4, 5, 20 and 21 are for the MPC56x (561-6) only. If you select one of these modules and MPC555 is the processor selected in the Resource Configuration block, then an error will be thrown on updating the model.

Edit period registers manually

When this option is selected, the **Clock prescaler field of MPWMSM Status/Control Register** and **Number of clock ticks per period** edit fields are activated. You can then set the PWM period by setting these values.

When this option is not selected, use the **Ideal period (sec)** field to set the PWM period parameters.

Ideal period (sec)

Specifies the desired period of the pulse signal. The mask then solves for the clock prescaler and the pulse period.

Initial duty cycle

Enter an initial value for the duty cycle (0 <= duty cycle <= 1). In setting this, the initial value is always known.

Clock prescaler field of MPWMSM Status/Control Register

Divides the counter clock to get the clock signal used to drive the PWM output. Note that the counter clock itself is derived from the MPC555 system clock as configured by the MPC555 Resource Configuration block (see MPC5xx MPC555 Resource Configuration).

Number of clock ticks per period

Determines the number of PWM counter ticks per pulse period. Valid values are `1 - 65535`.

Sample time

Sample time of the block.

Invert output polarity

Switches the output level for logic one and zero.

Activate transparent mode

Bypasses the register double buffers. When transparent mode is active, a software write to the Next Pulse Width Register is immediately transferred to the Pulse Width Register. When transparent mode is inactive, the updated value only takes effect at the start of the next period.

Hold output when at debug break point (freeze enable)

Stops the PWM counters when a breakpoint is hit while debugging, and holds the current output values.

MPC5xx MIOS Waveform Measurement

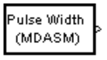
Purpose

Measure pulse width and pulse period measurement via MIOS Double Action Submodule (MDASM)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Modular Input/Output System (MIOS1)

Description



MIOS Waveform Measurement

Waveform measurement is a feature of the MIOS Double Action Submodule (MDASM) on the MPC555. The MIOS Waveform Measurement block currently implements the following features of the MDASM:

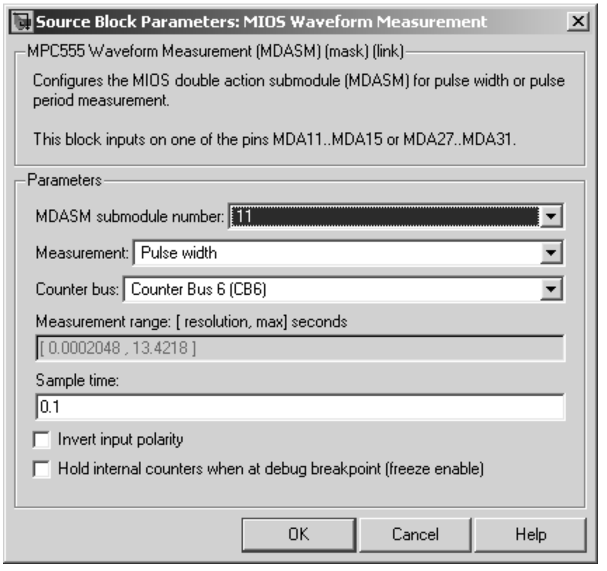
- *Pulse width measurement*: the MIOS Waveform Measurement block outputs the time from the leading edge of a pulse to the trailing edge of the same pulse.
- *Pulse period measurement*: the MIOS Waveform Measurement block outputs the time from the leading edge of a pulse to the next leading edge of a pulse.

Note that the minimum and maximum measurable pulse periods and pulse widths are dependent on the selected clock sources and their configurations.

You must configure the clock sources via the MPC555 Resource Configuration object. There are only two clock sources (assigned via the **Counter bus** parameter) assignable to the 10 MDASM modules. More than one MDASM can be assigned to a single clock source.

Refer to section 15.11, "MIOS Double Action Submodule (MDASM) Registers" in the *MPC555 User's Manual* for further information on the parameters described below.

Dialog Box



MDASM submodule number
: Select one of the 10 MIOS Double Action Submodules (MDASM) in the MPC555.

Measurement
: Select the mode of operation of the block: either pulse width measurement or pulse period measurement.

Counter bus
: Select one of the two counters that can be used as sources to drive the MDASM module. The counters must be configured via the MPC555 Resource Configuration object. See "MIOS1 Configuration Parameters" on page 5-764.

Measurement range: [resolution, max] seconds
: This read only field displays the measurement range of the pulse width or pulse period. The example shown is from the MPC555 real-time I/O demo model `mpc555rt_io`.

Sample time

The period at which Simulink reads the pulse width or period. The measurements are performed in hardware so it is not necessary to set the sample time to suit the expected period of the incoming signal.

Invert input polarity

Changes the sense of the leading edge of the pulse. When **Invert output polarity** is selected, the leading edge is rising. Otherwise, the leading edge is falling.

Hold internal counters when at debug break point (freeze enable)

Stops the clocks of the MDASM module when a breakpoint is hit while debugging.

Purpose

Provide CAN interface to execution profiling engine via CAN channel A

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Execution Profiling

Description

MPC555 Execution Profiling via CAN A

Provides a CAN interface to the execution profiling engine. On receipt of a start command message, logging of execution profile data is commenced. On completion of a logging run, the recorded data is automatically returned via CAN. You must specify the message identifiers for the start command and the returned data. These identifiers must be compatible with the values used by the host-side part of the execution profiling utility. See also MATLAB command `profile_mpc555`.

`profile_mpc555(connection)` collects and displays execution profiling data from an MPC555 target microcontroller that is running a suitably configured application generated by the Embedded Coder product. Set `connection` to `'CAN'` in order to collect data via a CAN connection between the target and the host computer. To use the CAN connection, you must have suitable CAN hardware installed on the host computer. This function will test for availability of CanCardX 1 or CanAc2Pci1 and defaults to a bit rate of 500k bits per second. If you need to use a different configuration, you should make a copy of this file (with a different name) and change the configuration data as required. The data collected is unpacked then displayed in a summary HTML report and as MATLAB graphic.

```
profdata = profile_mpc555(connection)
```

returns the execution profiling data in the format documented by `exprofile_unpack`.

See “The Profiling Command” for instructions for setting the bit rate automatically or manually.

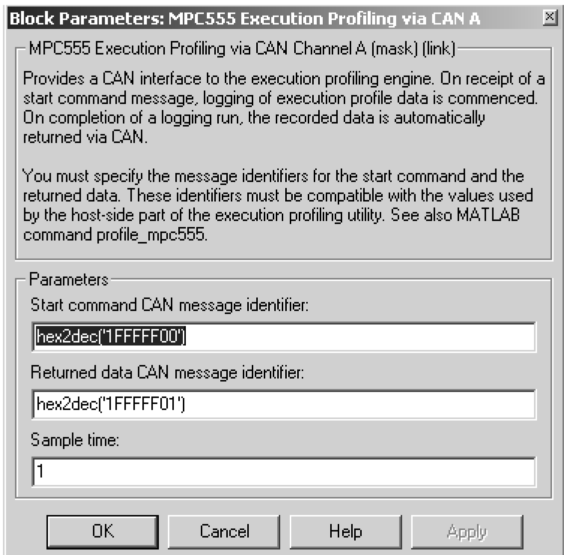
To configure a model for use with execution profiling, you must perform the following steps:

1 Make sure the model includes an MPC555 Execution Profiling block that provides an interface between the target-side profiling engine, and the host-side computer from which this command is run.

2 Make sure the execution profiling option is selected in the MPC5xx Options pane of the Configuration Parameters dialog box.

For more information see “Execution Profiling” which includes links to instructions for the example demo `mpc555rt_multitasking.mdl`.

Dialog Box



Start command CAN message identifier

Set the identifier of the message to start logging execution profiling data. You should use the default unless you have modified `profile_mpc555`. This identifier must be compatible with the values used by the host-side part of the execution profiling utility (`profile_mpc555`).

The utility profile_mpc555 provides a mechanism for initiating an execution profiling run and for uploading the recorded data to the host machine. To perform this procedure using a CAN connection between host and target, profile_mpc555 first sends a CAN message that is a command to start an execution profiling run. The CAN identifier for this message must be specified as the same value on the target as on the host. The host-side values are hard-coded in profile_mpc555. If you are using an un-modified version of the host side utility, you should use the default value for this CAN message identifier. These are visible to help you avoid using the same identifier for other tasks.

Returned data CAN message identifier
Set the message identifier for the returned data. As with the message identifier for the start command, the value specified here must be the same as the hard-coded value in profile_mpc555.

Sample time
The sample time of the block. The faster the sample time of the block, the faster data will be uploaded at the end of the execution profiling run. You may want to run this block slower than the fastest rate in the system because the execution profiling itself imposes some loading on the processor. You can minimize this extra loading by not running it at the fastest rate.

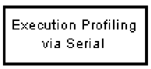
Purpose

Provide serial interface to execution profiling engine

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Execution Profiling

Description



MPC555 Execution Profiling via SCI1

Provides a CAN interface to the execution profiling engine. On receipt of a start command message, logging of execution profile data is commenced. On completion of a logging run, the recorded data is automatically returned via serial.

`profile_mpc555(connection)` collects and displays execution profiling data from an MPC555 target microcontroller that is running a suitably configured application generated by the Embedded Coder product. The `connection` may be set to '`serial`' in order to collect data via a serial connection between the target and the host computer.

The data collected is unpacked then displayed in a summary HTML report and as MATLAB graphic.

```
profdata = profile_mpc555(connection)
```

returns the execution profiling data in the format documented by `exprofile_unpack`.

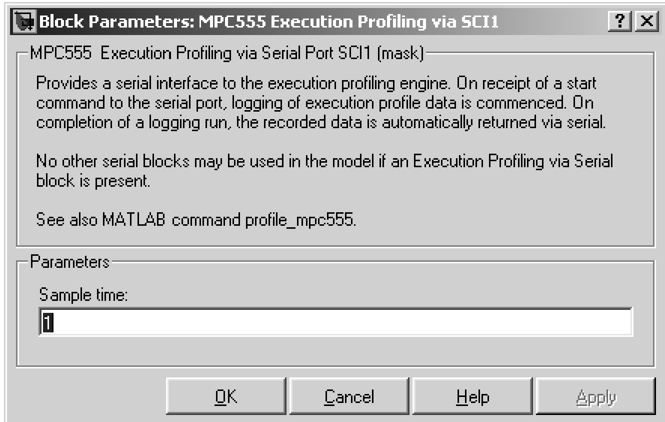
See "The Profiling Command" for instructions for setting the bit rate automatically or manually.

To configure a model for use with execution profiling, you must perform the following steps:

1. Make sure the model includes an MPC555 Execution Profiling block that provides an interface between the target-side profiling engine, and the host-side computer from which this command is run.
2. Make sure the execution profiling option is selected in the MPC5xx Options pane of the Configuration Parameters dialog box.

For more information see "Execution Profiling" which includes instructions for the example demo `mpc555rt_multitasking.mdl`.

Dialog Box



Sample time

The sample time of the block. The faster the sample time of the block, the faster data will be uploaded at the end of the execution profiling run. You may want to run this block slower than the fastest rate in the system because the execution profiling itself imposes some loading on the processor. You can minimize this extra loading by not running it at the fastest rate.

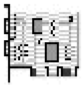
Purpose

Support device configuration for MPC5xx CPU and MIOS, QADC, and TouCAN submodules

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx

Description



MPC555 Resource Configuration

The MPC555 Resource Configuration block differs in function and behavior from conventional blocks. Therefore, we refer to this block as the MPC555 Resource Configuration *object*.

The MPC555 Resource Configuration object maintains configuration settings that apply to the MPC555 CPU and its MIOS, QADC, and TouCAN subsystems. Although the MPC555 Resource Configuration object resembles a conventional block in appearance, it is not connected to other blocks via input or output ports. This is because the purpose of the MPC555 Resource Configuration object is to provide information to other blocks in the model. MPC555 device driver blocks register their presence with the MPC555 Resource Configuration object when they are added to a model or subsystem; they can then query the MPC555 Resource Configuration object for required information.

To install a MPC555 Resource Configuration object in a model or subsystem, open the top-level Embedded Coder library and select the MPC555 Resource Configuration icon. Then drag and drop it into your model or subsystem, like a conventional block.

Having installed a MPC555 Resource Configuration object into your model or subsystem, you can then select and edit configuration settings in the MPC555 Resource Configuration window. See "Using the MPC555 Resource Configuration Window" on page 5-757 for further information.

Note Any model or subsystem using device driver blocks from the Embedded Coder library *must* contain an MPC555 Resource Configuration object. You should place an MPC555 Resource Configuration object at the top level system for which you are going to generate code. If your whole model is going to run on the target processor, put the MPC555 Resource Configuration object at the root level of the model. If you are going to generate code from separate subsystems (to run specific subsystems on the target), place an MPC555 Resource Configuration object at the top level of each subsystem. You should not have more than one MPC555 Resource Configuration object in the same branch of the model hierarchy. Errors will result if these conditions are not met.

When the MPC555 Resource Configuration block is placed into a model, it modifies the `preloadfcn` callback of the model. If you wish to add a command to the `preloadfcn` callback of a model that already has an MPC555 Resource Configuration block, do not remove the commands that are already installed.

Instead, copy the installed `preloadfcn` callback and append your commands. Then set the `preloadfcn` to the merged command. If you corrupt the `preloadfcn`, you can retrieve the command from any model that has an MPC555 Resource Configuration block, as the `preloadfcn` will be the same for all models. You can retrieve the `preloadfcn` with the following command:

```
plf = get_param(bdroot,'preloadfcn')
```

Types of Configurations

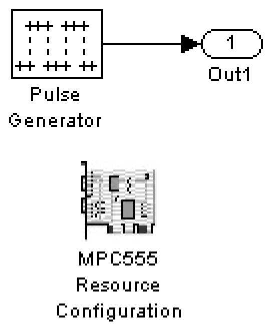
A *configuration* is a collection of parameter values affecting the operation of a group of device driver blocks in one of the Embedded Coder libraries, such as the MIOS1, QADC64 or TouCAN libraries. The MPC555 Resource Configuration object currently supports the following types of configurations:

- "System Configuration Parameters" on page 5-759: MPC555 clocks and other CPU-related parameters.
- "QADC64 Configuration Parameters" on page 5-761: parameters related to the Queued Analog-to-Digital Converter module (QADC).
- "QADC64E Configuration Parameters" on page 5-763: parameters related to the QADC for the MPC565.
- "MIOS1 Configuration Parameters" on page 5-764: parameters related to the Modular Input/Output System (MIOS).
- "TouCAN Configuration Parameters" on page 5-766: parameters related to the CAN 2.0B Controller Module (TouCAN).
- "Time Processor Unit (TPU3) Configuration Parameters" on page 5-769: parameters related to the Time Processor Unit module.
- "Serial Communications Interface (SCI) Configuration Parameters" on page 5-773: parameters related to the Serial Communications Interface.

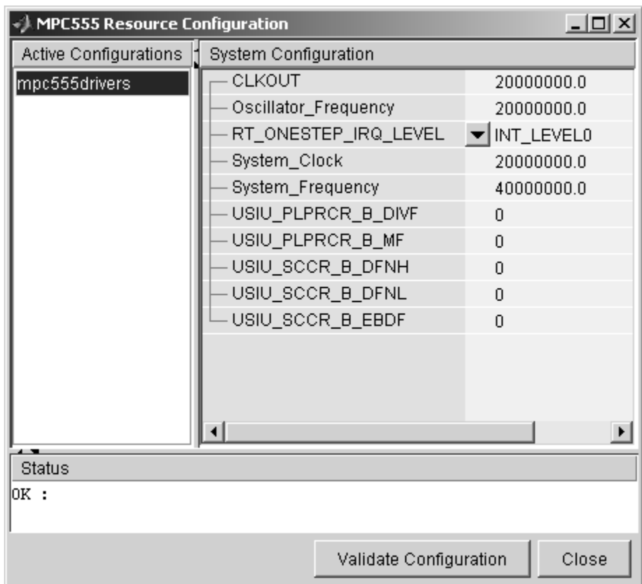
Active and Inactive Configurations

An *active* configuration is a configuration associated with blocks of the model or subsystem in which the MPC555 Resource Configuration object is installed. There is always an active MPC555 configuration. For any other configuration type (e.g., QADC, MIOS, or TouCAN), there is at most one active configuration. Such configurations are only active when relevant device driver blocks are present in the model or subsystem.

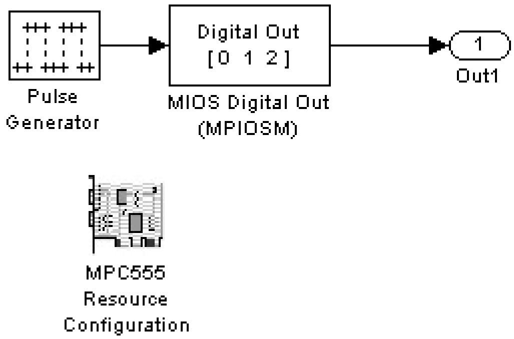
Consider this model, which contains a MPC555 Resource Configuration object but no MPC555 device driver blocks.



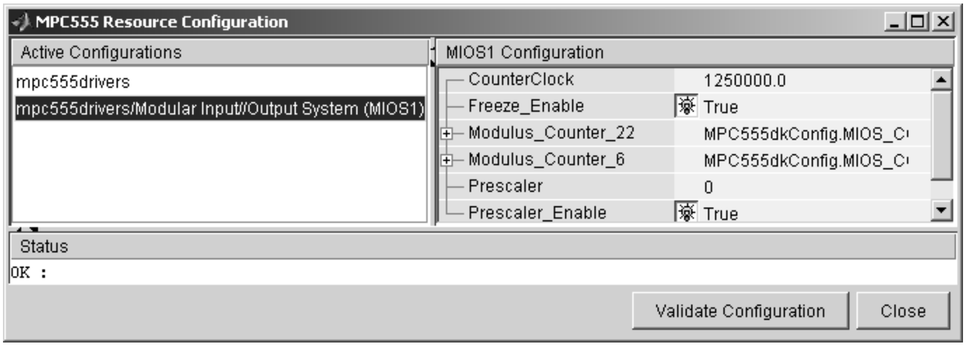
This model has only one active configuration, for the MPC555 itself, as shown in the MPC555 Resource Configuration window.



When a device driver block is added to the model, an appropriate configuration is created and activated. The following figure shows an MIOS Digital Out block added to the model.



The addition of the MIOS Digital Out block causes an MIOS configuration to be added to the list of active configurations, as shown in this figure.

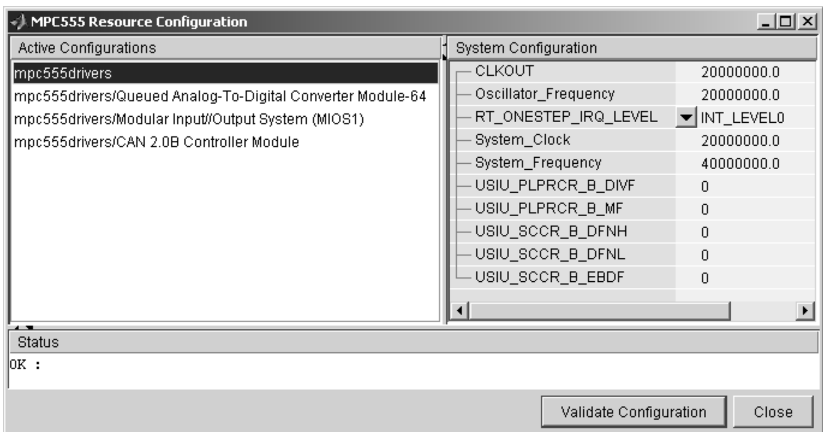


A configuration remains active until all blocks associated with it are removed from the model or subsystem. At that point, the configuration is in an *inactive* state. Inactive configurations are not shown in the MPC555 Resource Configuration window. You can reactivate a configuration by simply adding an appropriate block into the model.

Note When using device driver blocks from the Embedded Coder libraries in conjunction with the MPC555 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the MPC555 Resource Configuration block will operate incorrectly.

Using the MPC555 Resource Configuration Window

To open the **MPC555 Resource Configuration** window, install a MPC555 Resource Configuration object in your model or subsystem, and double-click on the MPC555 Resource Configuration icon. The **MPC555 Resource Configuration** window then opens.



MPC555 Resource Configuration Window

This figure shows the MPC555 Resource Configuration window for a model that has active configurations for MPC555, MIOS1, QADC, and TouCAN.

The MPC555 Resource Configuration window consists of the following elements:

- **Active Configurations** panel: This panel displays a list of currently active configurations. To edit a configuration, click on its entry in the list. The parameters for the selected configuration then appear in the **System configuration** panel.

  To link back to the library associated with an active configuration, right-click on its entry in the list. From the pop-up menu that appears, select **Go to library**.

  To see documentation associated with an active configuration, right-click on its entry in the list. From the popup menu that appears, select **Help**.

- **System configuration** panel: This panel lets you edit the parameters of the selected configuration. The parameters of each configuration type are detailed in "MPC555 Resource Configuration Window Parameters" on page 5-759.

Note There is no **Apply** or **Undo** functionality in the **System configuration** panel. All parameter changes are applied immediately.

- **Status** panel: The **Status** panel displays error messages that may arise if resource allocation conflicts are detected in the configuration.

- **Validate Configuration** button: After you edit a configuration, you should always click the **Validate Configuration** button to check for resource allocation conflicts. For example, if both TouCAN modules A and B are assigned to interrupt level IRQ 1, the **Validate Configuration** process will detect the conflict and display a warning in the **Status** panel.

  Note that the **Validate Configuration** button does not validate the entire model; it only checks for resource allocation conflicts related to the selected configuration. To detect problems related to the model as a whole, select **Update diagram** (**Ctrl+D**) from the Simulink Edit menu.

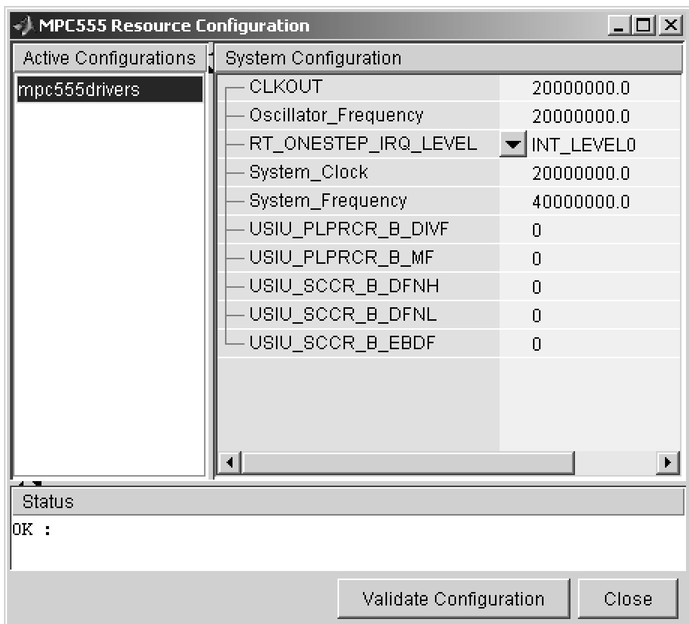
- **Close** button: Dismisses the window.

MPC555 Resource Configuration Window Parameters

The sections below describe the parameters for each type of configuration in the MPC555 Resource Configuration window. The default parameter settings are optimal for most purposes. If you want to change the settings, read the sections of the *MPC555 User's Manual* referenced below. You can find this document at the following URL:

```
http://www.freescale.com/files/microcontrollers/doc/user_guide/MPC555UM.pdf
```

System Configuration Parameters



RT_ONESTEP_IRQ_LEVEL

The `rt_OneStep` function is the basic execution driver of all programs generated by the Embedded Coder product. `rt_OneStep` is installed as a timer interrupt service routine; it sequences calls to the *`model`*`_step` function. The **RT_ONESTEP_IRQ_LEVEL** parameter lets you associate `rt_OneStep` with any of the available IRQ levels `(0..7)`. Do not select `Interrupts Disabled`, or the model will not work.

See the "Data Structures and Program Execution" section in the Embedded Coder documentation for a detailed description of the `rt_OneStep` function.

System Clock and Related Parameters

The parameters Oscillator_Frequency, `USIU_PLPRCR_B_DIVF`, `USIU_PLPRCR_B_MF`, `USIU_SCCR_B_DFNH`, `USIU_SCCR_B_DFNL`, `USIU_SCCR_B_EBDF` in the MPC555 group control the speed of the main clocks in the MPC555. Refer to section 8, "Clocks and Power Control," in the MPC555 User's Manual for information on these settings.

Some pre-defined configurations may be applied by inserting the block Switch Target Hardware Configuration into your model. This block is found in the Utilities sublibrary of the MPC555 Driver Library, see MPC5xx Switch Target Configuration. Insert this block in your model, then double-click on the block to choose a configuration from the available list. When one of the pre-defined configurations is selected, the appropriate settings will be applied automatically.

Note the Embedded Coder product only supports an `Oscillator_Frequency` of `4` MHz or `20` MHz; the setting of this parameter must correspond to the crystal frequency on your target hardware.

You might want to change these parameters in order to allow a different system clock value to be used; a faster system clock will increase the processing performance, as well as increasing power consumption. With default settings, the default values result in a system clock of 20 MHz for the MPC555. To gain additional processing power it may be desirable to increase the system clock. For the MPC555, the system clock may be increased up to 40

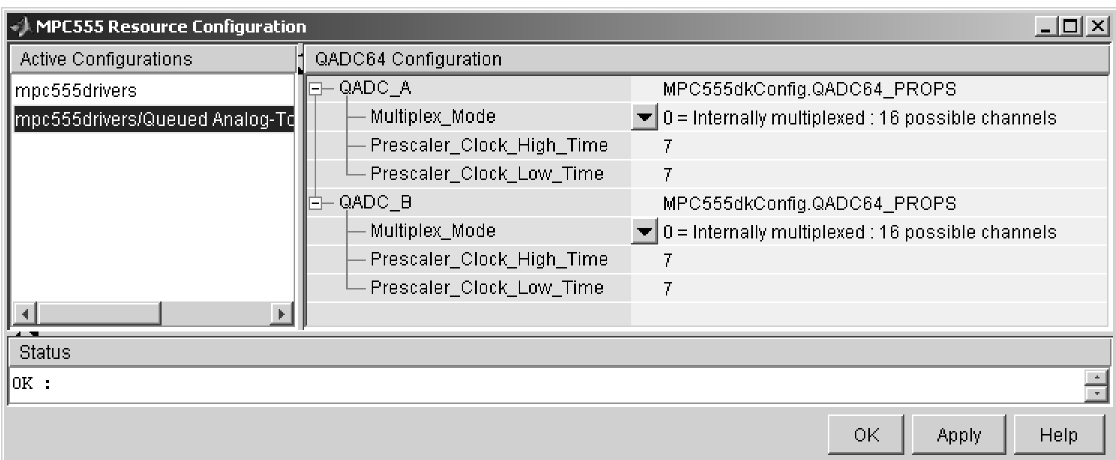
MHz. The exact settings that are required to achieve a desired system clock value may be calculated using the formulae provided in the MPC555 User Documentation. For example

System clock = `Oscillator_Frequency` * (MF+1) / (DIVF +1)

— where MF is the multiplying factor `USIU_PLPRCR_B_MF` and DIVF is the dividing factor `USIU_PLPRCR_B_DIVF`.

For example, if your hardware uses an external oscillator frequency of 20 MHz (e.g. as used on a phyCORE-MPC555 board), then changing the value of `USIU_PLPRCR_B_MF` from 0 to 1 will increase the system clock from 20 to 40 MHz. For different external oscillator frequencies or different processor variants you should consult the user documentation for your hardware.

QADC64 Configuration Parameters



The Queued Analog-To-Digital Converter Module 64 (QADC64) Configuration parameters configure the QADC64 operational mode and supports the blocks in the QADC sublibrary.

The QADC64 performs 10 bit analog to digital conversion on an input signal. Currently the blocks in this library support only the *continuous scan* mode of operation. In continuous scan mode, the QADC64 is set to run, and then continuously acquires data into its result buffer. Input is double buffered, so the model can read the result buffer at any time to get the latest available signal data.

The MPC555 has two QADC modules, QADC_A and QADC_B. You can program these individually. By default each QADC module has 16 input channels. By attaching an external multiplexer to three of the analog input pins, you can increase the number of possible channels to 41. These pins become outputs from the processor and can act as inputs to an analog multiplexer. The **Multiplex Mode** parameter determines whether the QADC64 operates in internally or externally multiplexed mode.

Refer to section 13, "Queued Analog-to-Digital Converter Module-64," in the *MPC555 User's Manual* for detailed information about the QADC64.

In general, you should not need to change any of the settings of the parameters described below from their defaults. The other parameters are advanced settings. Refer to section 13, "Queued Analog-to-Digital Converter Module-64," in the *MPC555 User's Manual* for information on these settings.

Multiplex Mode

Configures the QADC64 for internally or externally multiplexed mode by setting the MUX bit. The MUX bit determines the interpretation of the channel numbers and forces the MA[2:0] pins to be outputs. Valid settings are

- `0 = Internally multiplexed :  16 possible channels`
- `1 = Externally multiplexed :  41 possible channels`

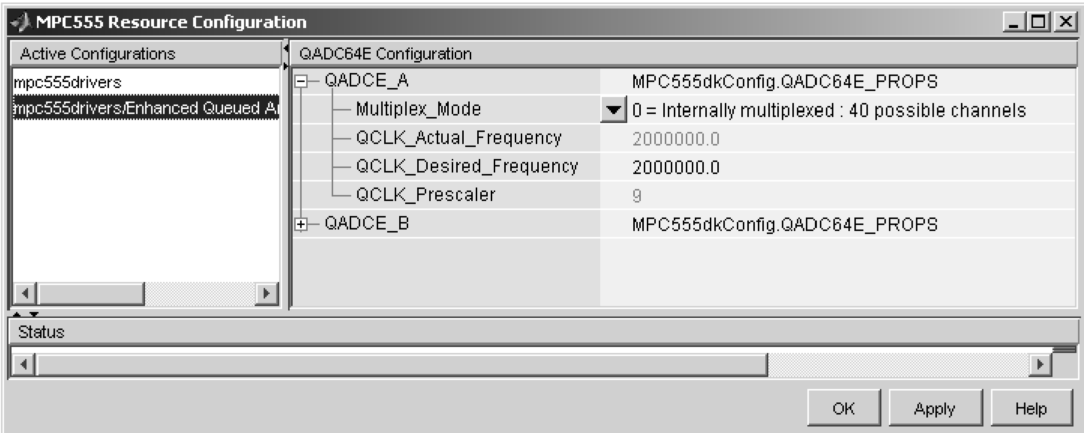
Prescaler Clock High Time

Prescaler clock high (PSH) time. The default is 7. The PSH field selects the QCLK high time in the prescaler. PSH value plus 1 represents the high time in IMB clocks.

Prescaler Clock Low Time

Prescaler clock low (PSL) time. The default is 7. The PSL field selects the QCLK low time in the prescaler. PSL value plus 1 represents the low time in IMB clocks.

QADC64E Configuration Parameters



The Enhanced QADC functions are for MPC56x processors – you will see an error message if you try to configure these for an MPC555. Use QADC blocks for an MPC555; for an MPC56x set your target processor accordingly in the Target Preferences and then you can use the QADCE blocks.

The Enhanced Queued Analog-To-Digital Converter Module 64 (QADC64E) Configuration parameters configure the QADC64E operational mode and supports the blocks in the Enhanced QADC sublibrary.

Multiplex Mode

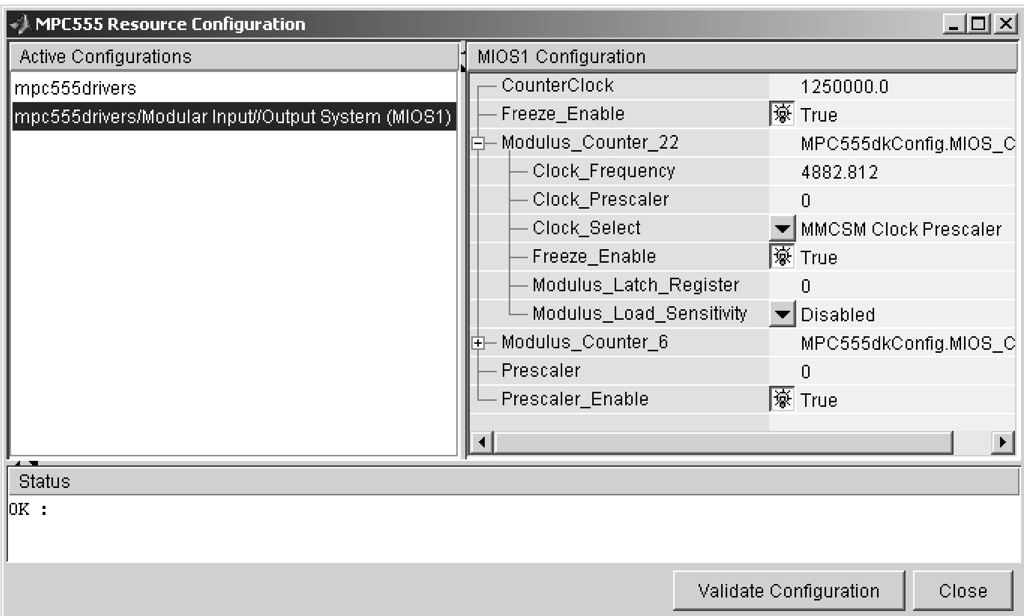
Configures the QADC64 for internally or externally multiplexed mode by setting the MUX bit. The MUX bit determines the interpretation of the channel numbers and forces the MA[2:0] pins to be outputs. Valid settings are

- `0 = Internally multiplexed :  40 possible channels`
- `1 = Externally multiplexed :  65 possible channels`

QCLK_Desired_Frequency

Set the Q clock frequency you want here. The `QCLK_Actual_Frequency` field displays the true value achieved. `QCLK_Actual_Frequency` and `QCLK_Prescalar` are read only fields for information.

MIOS1 Configuration Parameters



CounterClock

The MIOS counter clock is generated by the MIOS counter prescaler submodule. The MIOS counter clock drives the other MIOS1 submodules. The value shown for the counter clock is calculated automatically as the system clock frequency divided by the prescaler value.

Freeze Enable

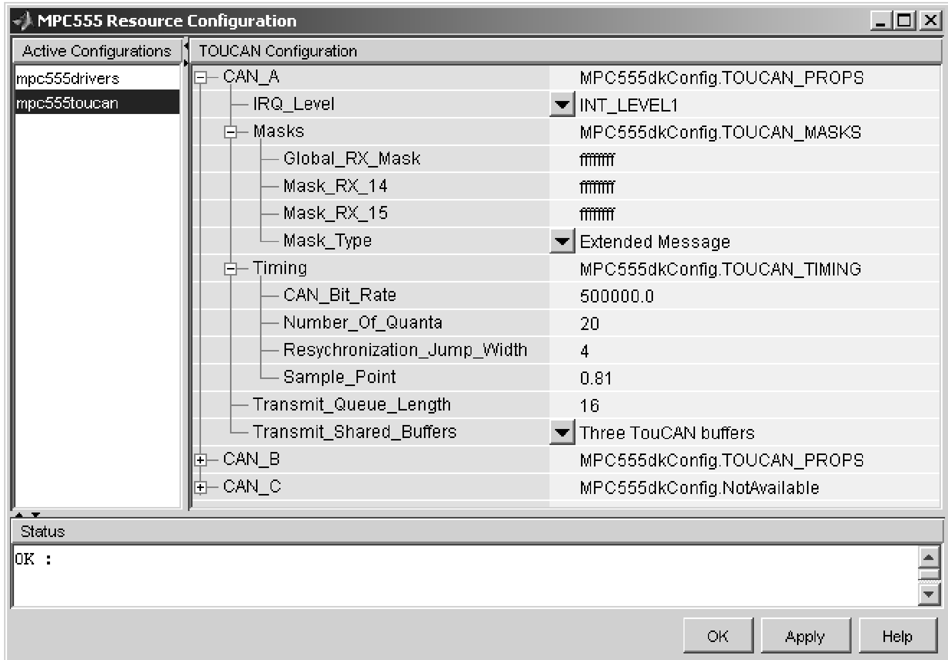
This allows all counters on the MIOS1 to be frozen when the processor is stopped while debugging. Note that this is in addition to the **Freeze Enable** setting for individual submodules on the MIOS1. To allow the counters on a particular submodule to be stopped, select Freeze enable here, and select **Hold output when at debug break point (freeze enable)** in the block parameters associated with the submodule (e.g., MIOS Pulse Width Modulation block or MIOS Waveform Measurement block).

Modulus Counter 6 and 22

These two counters provide reference clocks to submodules such as the MIOS Pulse Width Modulation Submodule and the MIOS Double Action Submodule (Frequency / Period measurement) subsystems. If you change the **Clock Select** to anything other than `MMCSM Clock Prescaler`, the MIOS Pulse Width Modulation and MIOS Waveform Measurement blocks will not work as expected. To change the clock frequency and hence the available resolution of pulse width modulation and waveform measurement, change the **Clock Prescaler** to a value between 0 and 255.

Refer to section 15.10, "MIOS Modulus Counter Submodule (MMCSM)," in the *MPC555 User's Manual* for information on these settings.

TouCAN Configuration Parameters



The parameters listed below are the same for TouCAN modules A and B (and C, for MPC56x). Consult Section 16 of the *MPC555 User's Manual* before editing the TouCAN configuration parameter defaults.

IRQ Level

The transmit queue for each TouCAN module requires a processor interrupt to run. Select an interrupt level (0-31) that is not used by any other device. Use the **Apply** button to make sure you do not select an interrupt level that is already in use. Do not disable interrupts: this will stop the TouCAN Transmit block from working correctly.

Mask Configuration Parameters

Global RX Mask

Buffers 0-13 use this mask. Setting a bit to 0 in the mask causes the corresponding bits in the incoming message's identifier to be masked out (i.e., ignored).

`0` – Corresponding bit in the incoming message's identifier is "don't care"

`1` – Corresponding bit in the incoming message's identifier is checked against the identifier specified in the TouCAN Receive block associated with this buffer.

Mask RX 14

Same as **Global RX Mask**, but the mask applies only to buffer 14.

Mask RX 15

Same as **Global RX Mask**, but the mask applies only to buffer 15.

Mask Type

Specify whether the buffer masks are Standard or Extended frame IDs. If you want to receive Extended Frames in your model, you should set the **Mask Type** to **Extended Message**. The mask type option tells the compiler how to map the bits specified in the mask options to the bits in the hardware. The decision as to whether a message is a Standard or Extended frame is defined on a per message buffer basis.

Timing Configuration Parameters

CAN Bit Rate

Enter the desired bit rate. The default bit rate is `500000.0`.

Number of Quanta

The number of TouCAN clock ticks per message bit.

Resynchronization Jump Width

The maximum number of clock ticks that the TouCAN device can resynchronize over when it detects that it is losing message synchronization.

Sample Point

The point in the message where the TouCAN tries to sample the value of the message bit, between 0 and 1.

Slew Rate

You cannot select the slew rate for the TouCAN modules. By default, the slow slew rate is selected for the TouCAN modules. This results in a slew rate of 50ns for TouCAN C, and 200ns for the other modules.

Transmission Configuration Parameters

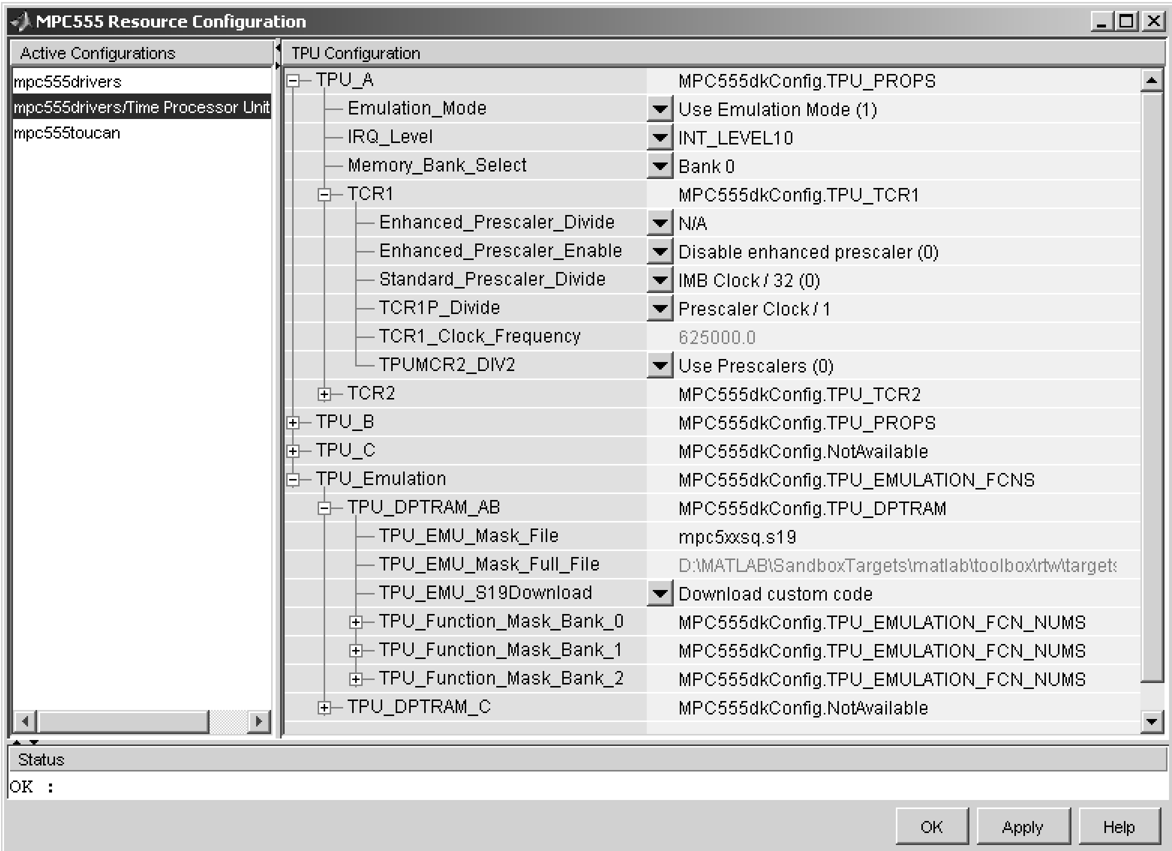
Transmit Queue Length

Length (number of messages) of the transmit queue. The transmit queue holds messages that are waiting to be transmitted. An increase in performance can be achieved by reducing the queue length. However, if the queue's length is too small it may become full, causing messages to be lost.

Transmit Shared Buffers

Choose either `Single TouCAN Buffer` or `Three TouCAN Buffers`. This parameter is used in conjunction with all TouCAN Transmit blocks in the model for this TouCAN module that are operating in `Queued transmission with shared buffer` mode. If you select `Single TouCAN Buffer`, then all messages that are queued will be transmitted via a single hardware buffer; in this case, it is possible that a low priority message in the transmit buffer will block higher priority messages that are in the queue. To avoid this problem, use the option `Three TouCAN Buffers`. When three buffers are used, the message entered into arbitration to be transmitted via the CAN bus is always the highest priority message available; furthermore in this mode the TouCAN module is able to transmit messages continuously by re-loading hardware buffers that become empty while another buffer is active transmitting.

Time Processor Unit (TPU3) Configuration Parameters



Emulation_Mode

The default is to `Use ROM TPU Functions (0)`. Select `Use Emulation Mode (1)` to use downloaded TPU functions in DPTRAM. Use the parameters under **TPU_Emulation** to configure downloads for emulation mode. For an example see the demo model mpc555rt_tpu_emu. Note that CCP Program_Prepare downloads will fail because DPTRAM_AB contains TPU microcode for emulation mode.

IRQ_Level
This enables TPU interrupts. The default is disabled. If your model contains any TPU3 Programmable Time Accumulator blocks, you will need to choose an interrupt level.

Memory_Bank_Select
Select Bank 0, 1 or 2. If you select an invalid memory bank for the TPU module (e.g. `Bank 2` for TPU C) you will see an error message when you click **Apply**. This must match the selection for the parameters TPU_Function_Mask_Bank_0 (also Bank_1, Bank_2).

The TCR1 and TCR2 timebases are configurable for TPU Channels A, B and C.

TCR1
The parameters under the TCR1 tree allow you full control to specify the clock speed of the TCR1 timebase. Consult Section 17 of the *MPC555 User's Manual* before editing the TPU configuration parameter defaults. The parameters listed below are the same for TPU modules A, B and C.

Enhanced_Prescaler_Divide
If you choose to use the Enhanced_Prescaler_Divide, then you can choose to divide the IMB clock down by either 2, 4, 6, 8, ... , 60, 62, 64.

Enhanced_Prescaler_Enable
Here you can choose whether you use the Standard Prescaler (set by `Standard_Prescaler_Divide`) or the Enhanced Prescaler (set by `Enhanced_Prescaler_Divide`) to derive the Prescaler Clock.

Standard_Prescaler_Divide
If you choose to use the Standard_Prescaler_Divide then you can choose to divide the IMB clock down by either 32 or 4.

TCR1P_Divide
Whichever type of prescaler you choose (standard or extended), there is a further prescaler that is applied to the clock.

TCR1P_Divide divides the Prescaler Clock by 1, 2, 4, or 8. The resulting clock is the TCR1 timebase.

TCR1_Clock_Frequency
Read-only field displaying calculated TCR1 clock frequency.

TPUMCR2_DIV2
TPUMCR2_DIV2 (the last setting under the tree) allows you to choose to use a set of prescalers to divide the IMB clock down further (`Use Prescalers (0)`), or to just divide the IMB clock by two (`IMB Clock / 2 (1)`). If you choose the divide by two option then none of the other settings are applicable and are marked `N/A`. Note this is the last setting purely because the parameters are laid out in alphabetical order.

TCR2
The parameters under the TCR2 tree for specifying the clock speed of the TCR2 timebase are the same for TPU modules A, B and C. You can configure the TCR2 to use an external clock.

TCR2P_Divide
You can choose to divide the TCR2 prescaler clock down by either 1, 2, 4, or 8.

TCR2_Clock_Frequency
Read-only field displaying calculated TCR2 clock frequency when using the gated IMB clock. This field displays zero when using an external clock, as it cannot predict an external clock signal.

TCR2_Counter_Clock_Source
Select from `Rise transition T2CLK`, `Gated IMB clock`, `Fall transition T2CLK`, or `Rise & fall transition T2CLK`.

The `Gated IMB` clock setting uses the T2CLK pin to gate the internal clock as a source for TCR2 (a logical AND between the input on the T2CLK pin and the IMB clock is performed).

The other settings allow TCR2 to be clocked from the selected edge of an external clock signal applied to the T2CLK pin.

TCR2_PSCK2

See the *MPC555 User's Manual* for the effects of setting the TCR2_PSCK2 bit. The default, `Divide by 1`, leaves the **TCR2P_Divide** setting the only prescaler applied to the clock (if using an external clock). If using the gated IMB clock there is always an additional implicit divide by 8.

TPU_Emulation

Use these settings to configure downloads for TPU emulation mode.

TPU_DPTRAM_AB and TPU_DPTRAM_C

Use the settings under these two parameters to configure emulation mode for TPU modules A and B **(TPU_DPTRAM_AB)** and/or TPU modules C (**TPU_DPTRAM_C**). The parameters listed below are the same for TPU modules A, B and C.

TPU_EMU_Mask_File

Enter the name of the S19 file containing the TPU functions to be downloaded. The specified file must be either in the current working folder OR the MATLAB path if an absolute path is not explicitly specified. Note the file name will not be accepted unless **TPU_EMU_S19Download** is set to `Download custom code`. This parameter retains a memory of the last file specified.

The S19 file must be produced from an `.asc` microcode mask file and a TPU microcode assembler. The TPU function names and TPU function numbers are specified in the `.asc` file. Make sure you enter the same TPU function names and numbers in the **TPU_Function_Mask_Bank** parameters.

TPU_EMU_Mask_Full_File

Read only field displaying the full path to the download file. Check to see if the correct file is shown.

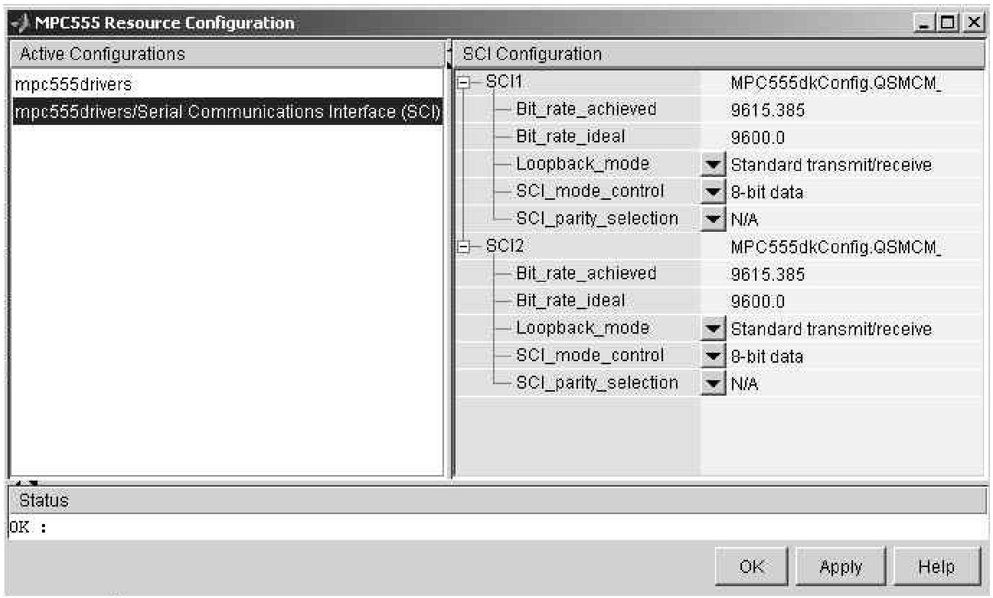
TPU_EMU_S19Download

Select `Download custom code` to download to DPTRAM for emulation mode. The default is `No code download`.

TPU_Function_Mask_Bank_0 (also Bank_1, Bank_2)

Use the parameters under here to specify which TPU Function Numbers correspond to which TPU functions. For example, typing `PTA` for `TPU_Function_D` will specify that the PTA function is configured as TPU function number 13. If you enter a string that is not a valid TPU function name, when you click **Apply** an error message appears in the status field, followed by a list of possible TPU Function Names and their corresponding full function names. Names must be exact including case. The specified TPU function names and numbers must correspond to those specified in the **TPU_EMU_Mask_File**.

Serial Communications Interface (SCI) Configuration Parameters



Bit_rate_achieved
: This read-only field shows the achieved serial interface bit rate. In general this value differs slightly from the requested bit rate, but is the closest value that can be achieved by setting allowed values in the MPC555 registers SCC1R0 and SCC2R0 for QSMCM submodules SCI1 and SCI2 respectively.

Bit_rate_ideal
: Enter the desired bit rate for serial communications in this field. Appropriate register settings will be calculated automatically. You can check the actual bit rate in the **Bit_rate_achieved** field.

Loopback_mode_enable
: Select either `Standard transmit/receive` or `Loopback mode enabled`. The loopback mode may be useful for test purposes where the serial interface is required to receive data that it transmitted itself.

SCI_mode_control
: Select the desired combination of word length and parity/no parity.

Parity_selection
: If parity is enabled, you must select `Odd parity` or `Even parity`.

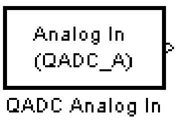
Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Queued Analog-To-Digital Converter Module-64

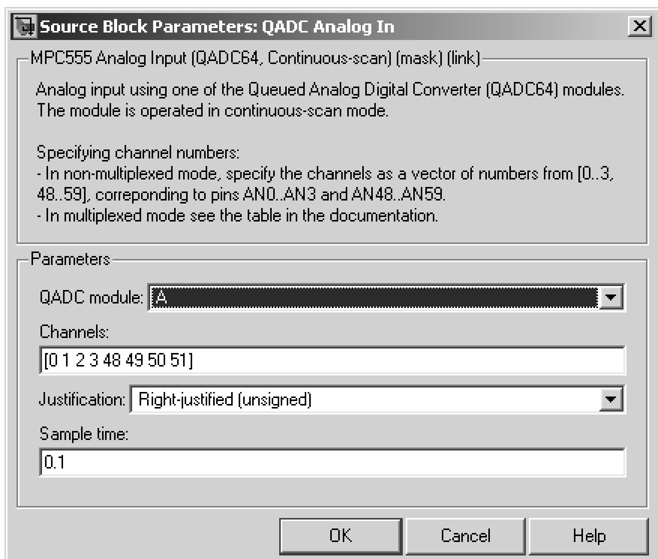
Description



The QADC Analog In block sets the QADC64 into continuous scan mode. It then samples the specified channels at the specified rate. In continuous scan mode, the analog-to-digital converter is scanned as fast as possible, at a rate much faster than the sample rate of the model. Using continuous scan mode your application obtains the latest signal value.

The MPC555 has two QADC modules, A and B. You can program these individually. You can place only one instance of the QADC Analog In block per module in your model or subsystem.

Dialog Box



QADC module
Select module A or B.

Channels
A vector of numbers representing channels to be scanned. See "Channel Number Selection" on page 5-776 below.

Justification
Converted data is read from the 10-bit wide QADC64 result word table into a 16-bit word. Data from the result word table can be accessed in three different formats. The **Justification** menu selects from the following formats:

- `Right-justified (unsigned)`: with zeros in the higher order unused bits.
- `Left-justified (signed)`: with the most significant bit inverted to form a sign bit, and zeros in the unused lower order bits. In this mode, zero is treated as the half scale of the input range.
- `Left-justified (unsigned)`: with zeros in the unused lower order bits.

Refer to section 13.13, in the "Queued Analog-to-Digital Converter Module-64" section of the *MPC555 User's Manual* for further information.

Sample time
Block sample time; determines sample rate at which the port is monitored.

Channel Number Selection

A channel number in the **Channels** vector selects the input channel number corresponding to the analog input pin to be sampled and converted. The analog input pin channel number assignments and the pin definitions vary, depending on whether the QADC64 is operating in multiplexed or nonmultiplexed mode. The queue scan mechanism makes no distinction between an internally or externally multiplexed analog input.

The following two tables show the mapping between the channel numbers and the hardware pins for the two scanning modes (multiplexed and nonmultiplexed).

For example, in nonmultiplexed mode, to scan all 16 channels of the QADC64 you would specify the following vector in the **Channels** field:

```
[ 0 1 2 3 48 49 50 51 52 53 54 55 56 57 58 59 ]
```

Nonmultiplexed Scan Mode

Port Pin Name	Analog Pin Name	Pin Type (I/O)	Channel Number
PQB0	A_AD0 / AN0	I	0
PQB1	A_AD1 / AN1	I	1
PQB2	A_AD2 / AN2	I	2
PQB3	A_AD3 /AN3	I	3
PQB4	A_AD4 / AN48	I	48
PQB5	A_AD5 / AN49	I	49
PQB6	A_AD6 / AN50	I	50
PQB7	A_AD7 / AN51	I	51
PQA0	A_AD8 / AN52	I/O	52
PQA1	A_AD9 / AN53	I/O	53
PQA2	A_AD10 / AN54	I/O	54
PQA3	A_AD11 / AN55	I/O	55
PQA4	A_AD12 / AN56	I/O	56
PQA5	A_AD13 / AN57	I/O	57
PQA6	A_AD14 / AN58	I/O	58
PQA7	A_AD15 / AN59	I/O	59

Multiplexed Scan Mode

Port Pin Name	Analog Pin Name	Pin Type (I/O)	Channel Number
PQB0	A_AD0 / ANw	I	0–14 even
PQB1	A_AD1 / ANx	I	1–15 odd
PQB2	A_AD2 / ANy	I	16–30 even
PQB3	A_AD3 / ANz	I	17–31 odd
PQB4	A_AD4 / AN48	I	48
PQB5	A_AD5 / AN49	I	49
PQB6	A_AD6 / AN50	I	50
PQB7	A_AD7 / AN51	I	51
PQA3	A_AD11 / AN55	I/O	55
PQA4	A_AD12 / AN56	I/O	56
PQA5	A_AD13 / AN57	I/O	57
PQA6	A_AD14 / AN58	I/O	58
PQA7	A_AD15 / AN59	I/O	59

Note PQA0, PQA1 and PQA2 (corresponding to channels 52–54) are used as output pins (MA0, MA1, and MA2) to drive an external demultiplexer.

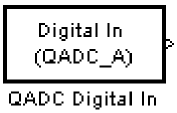
Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) pins as digital inputs

Library

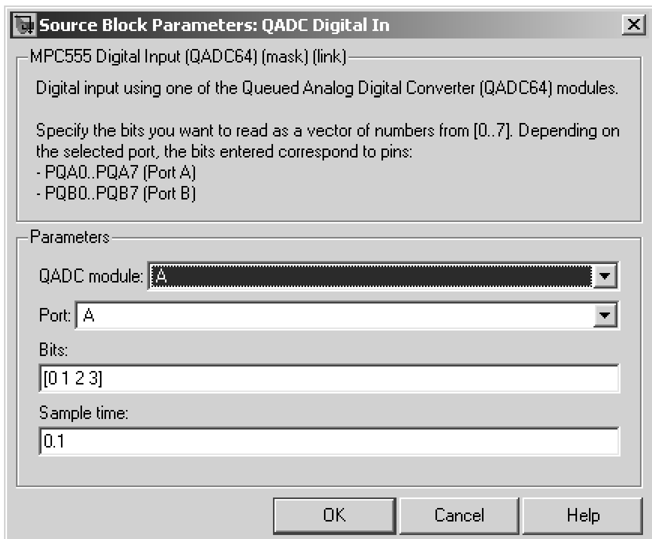
Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Queued Analog-To-Digital Converter Module-64

Description



The QADC Digital In block allows you to treat the QADC64 pins as digital inputs. Each QADC64 module has two 8-bit ports, A and B. You can use any bit on either port as a digital input.

Dialog Box



QADC module

Select module A or B.

Port

Select an 8 bit port (A or B) on the module.

Bits

A vector of bits (numbered 0-7) to read. The vector should not be longer than eight elements.

Sample time

Block sample time; determines sample rate at which the port is monitored.

Mapping Bits To Hardware Pins

Use this table to work out how the block ports and bits map to processor pins on the MPC555.

Relationship of Port/Bit Parameters to Hardware Pins

Port	Bit	Hardware Pin
B	0	A_AD0 / PQB0
B	1	A_AD1 / PQB1
B	2	A_AD2 / PQB2
B	3	A_AD3 / PQB3
B	4	A_AD4 / PQB4
B	5	A_AD5 / PQB5
B	6	A_AD6 / PQB6
B	7	A_AD7 / PQB7
A	0	A_AD8 / PQA0
A	1	A_AD9 / PQA1
A	2	A_AD10 / PQA2
A	3	A_AD11 / PQA3
A	4	A_AD12 / PQA4
A	5	A_AD13 / PQA5

Relationship of Port/Bit Parameters to Hardware Pins (Continued)

Port	Bit	Hardware Pin
A	6	A_AD14 / PQA6
A	7	A_AD15 / PQA7

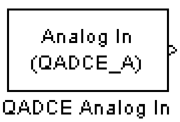
Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) in continuous scan mode on MPC56x (561-6)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Enhanced Queued Analog-To-Digital Converter Module-64

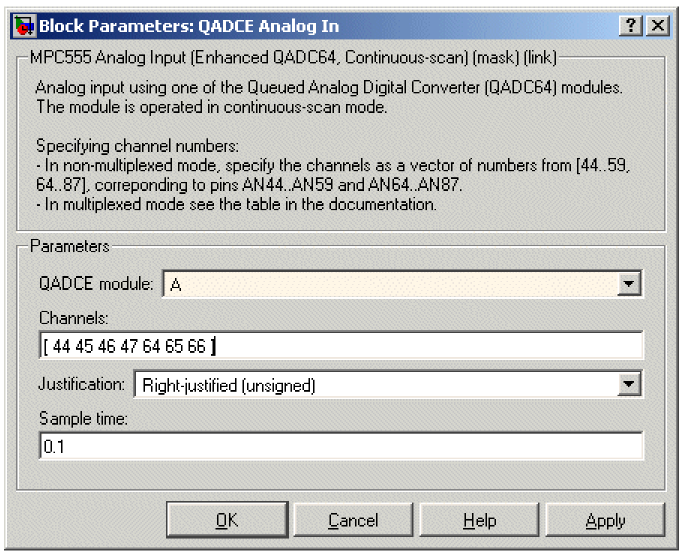
Description



The QADCE Analog In block sets the QADC64E into continuous scan mode. It then samples the specified channels at the specified rate. In continuous scan mode, the analog-to-digital converter is scanned as fast as possible, at a rate much faster than the sample rate of the model. Using continuous scan mode your application obtains the latest signal value.

The MPC56x has two QADC64E modules, A and B. You can program these individually. You can place only one instance of the QADCE Analog In block per module in your model or subsystem.

Dialog Box



QADC module

Select module A or B.

Channels

A vector of numbers representing channels to be scanned. A channel number in the **Channels** vector selects the input channel number corresponding to the analog input pin to be sampled and converted.

The analog input pin channel number assignments and the pin definitions vary, depending on whether the QADC64E is operating in multiplexed or nonmultiplexed mode. The queue scan mechanism makes no distinction between an internally or externally multiplexed analog input.

In nonmultiplexed mode, specify a vector of numbers from [44..59 64..87] corresponding to pins AN44..AN59 and AN64..AN87.

See the table following for the mapping in multiplexed mode between the channel numbers and the hardware pins.

Justification

Converted data is read from the 10-bit wide QADC64E result word table into a 16-bit word. Data from the result word table can be accessed in three different formats. The **Justification** menu selects from the following formats:

`Right-justified (unsigned)`: with zeros in the higher order unused bits.

`Left-justified (signed)`: with the most significant bit inverted to form a sign bit, and zeros in the unused lower order bits. In this mode, zero is treated as the half scale of the input range.

`Left-justified (unsigned)`: with zeros in the unused lower order bits.

Sample time

Block sample time; determines sample rate at which the port is monitored

Mapping Bits To Hardware Pins

Use the following table to work out how the block ports and bits map to processor pins on the MPC565 in multiplexed mode.

In summary

- No multiplexing:

  channels available 44-59 and 64-87
- A only multiplexing:

  channels available 0-31; 48-51; 55-59; 64-87
- B only multiplexing:

  channels available 0-31; 48-59; 64-71; 75-87
- A and B multiplexing:

  channels available 0-31; 48-51; 55-59; 64-71; 75-87

Multiplexed Scan Mode

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
ANw/A_PQB0	AN00 to AN07	-	Input	0 to 7
ANx/A_PQB1	AN08 to AN15	-	Input	8 to 15
ANy/A_PQB2	AN16 to AN23	-	Input	16 to 23
ANz/A_PQB3	AN24 to AN31	-	Input	24 to 31
A_PQB0	AN44	ANw	Input/Output	44
A_PQB1	AN45	ANx	Input/Output	45
A_PQB2	AN46	ANy	Input/Output	46
A_PQB3	AN47	ANz	Input/Output	47
A_PQB4	AN48	-	Input/Output	48

Multiplexed Scan Mode (Continued)

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
A_PQB5	AN49	-	Input/Output	49
A_PQB6	AN50	-	Input/Output	50
A_PQB7	AN51	-	Input/Output	51
A_PQA0	AN52	MA0	Input/Output	52
A_PQA1	AN53	MA1	Input/Output	53
A_PQA2	AN54	MA2	Input/Output	54
A_PQA3	AN55	-	Input/Output	55
A_PQA4	AN56	-	Input/Output	56
A_PQA5	AN57	-	Input/Output	57
A_PQA6	AN58	-	Input/Output	58
A_PQA7	AN59	-	Input/Output	59
B_PQB0	AN64	-	AMUX Input	64
B_PQB1	AN65	-	AMUX Input	65
B_PQB2	AN66	-	AMUX Input	66
B_PQB3	AN67	-	AMUX Input	67
B_PQB4	AN68	-	AMUX Input	68
B_PQB5	AN69	-	AMUX Input	69
B_PQB6	AN70	-	AMUX Input	70
B_PQB7	AN71	-	AMUX Input	71
B_PQA0	AN72	MA0	AMUX Input	72
B_PQA1	AN73	MA1	AMUX Input	73
B_PQA2	AN74	MA2	AMUX Input	74

Multiplexed Scan Mode (Continued)

Port Pin Name	Analog Pin Name	Other Functions	Pin Type (I/O)	Channel Number
B_PQA3	AN75	-	AMUX Input	75
B_PQA4	AN76	-	AMUX Input	76
A_PQA5	AN77	-	AMUX Input	77
A_PQA6	AN78	-	AMUX Input	78
A_PQA7	AN79	-	AMUX Input	79
-	AN80	-	-	80
-	AN81	-	-	81
-	AN82	-	-	82
-	AN83	-	-	83
-	AN84	-	-	84
-	AN85	-	-	85
-	AN86	-	-	86
-	AN87	-	-	87

In this table, MA0 to MA2 indicates these pins (A_ and B_PQA0-2) are used as output pins to drive an external demultiplexer.

Purpose

Input driver enables use of Queued Analog-Digital Converter (QADC64) pins as digital inputs on MPC56x (561-566)

Library

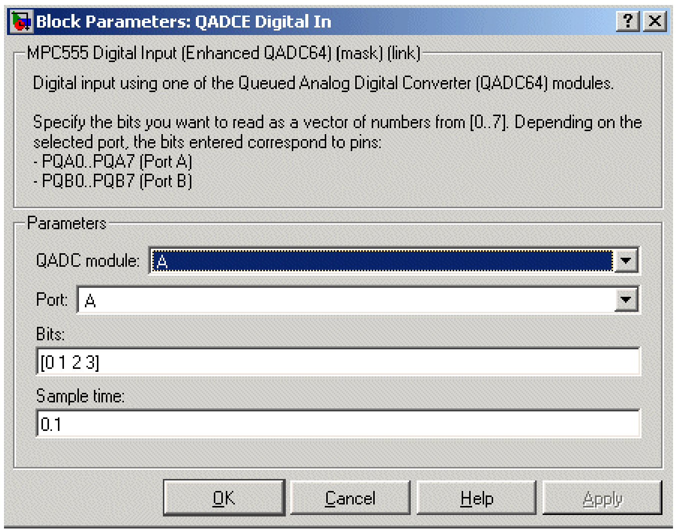
Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Enhanced Queued Analog-To-Digital Converter Module-64

Description

QADCE Digital In

The QADCE Digital In block allows you to treat the QADC64E pins as digital inputs. Each QADC64E module has two 8-bit ports, A and B. You can use any bit on either port as a digital input.

Dialog Box



QADC module

Select module A or B.

Port

Select an 8 bit port (A or B) on the module.

Bits

Specify a vector of bits (numbered 0-7) to read. The vector should not be longer than eight elements. Depending on the selected port, the bits entered correspond to pins PQA0 to PQA7 (port A) or PQB0 to PQB7.

Sample time

Block sample time; determines sample rate at which the port is monitored

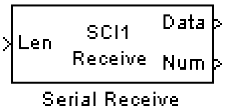
Purpose

Configure MPC555 for serial receive on either of QSMCM submodules SCI1 or SCI2

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Serial Communications Interface (SCI)

Description



The Serial Receive block receives bytes via either of the MPC555 QSMCM submodules SCI1 or SCI2. It requests either a fixed number of bytes to be received, or, by enabling the first input, a variable number of bytes can be requested each time this block is called. When the block is called, the requested number of bytes are retrieved from a hardware buffer provided by the submodule SCI1 or SCI2. On SCI1, the total size of the buffer is 16 bytes; note however that the effective capacity is reduced due to the hardware behavior and the circular mode of buffer operation used by the software driver. You should design your application on the basis of 9 bytes for the maximum buffer size for SCI1. On SCI2, the size of this buffer is 1 byte.

If the buffer contains fewer bytes than the number requested, these bytes are pulled from the buffer and made available at the block output. The number of bytes actually retrieved from the buffer is made available at the second output. This block will only retrieve bytes that have already been received and placed in the hardware buffer; it will never wait for additional data to be received.

To configure the serial interface bit rate and data format, see "Serial Communications Interface (SCI) Configuration Parameters" on page 5-773.

The device driver used for the Serial Receive block does not require the use of CPU interrupts.

Block Inputs and Outputs

The first input can be enabled so a variable number of bytes can be requested each time.

The second input, if enabled, is a reset signal, which must have a Boolean data type. You must reset the SCI1 module if an overrun error or framing or parity error occurs. No reset is required for SCI2.

The first output (marked `Data`) pulls bytes from the buffer — either the number requested or the number available, whichever is the lower. Note that the number requested is the value of the first input signal if supplied, or the width of the output signal otherwise.

The second output (marked `Num`) is the number of bytes actually retrieved from the buffer. Up to four outputs can be enabled — the third showing framing error and parity error flags, and the fourth showing overrun flags.

See "Data Type Support and Scaling for Device Driver Blocks" for information on supported input/output data types and scaling of input/output signals.

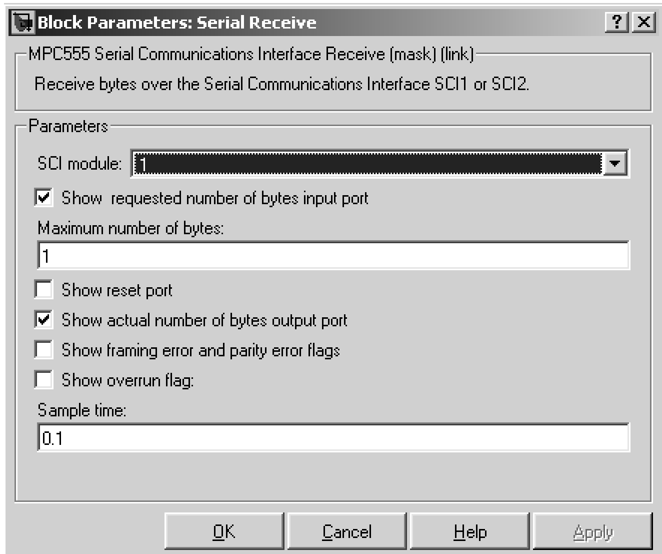
This block supports the following three serial data frame formats:

Frame size	Start bit	Data length	Parity/ Control bit	Stop bit
10–bit	1	7–bit	1	1
10–bit	1	8–bit	—	1
11–bit	1	8–bit	1	1

This block does not support the following two serial data frame formats:

Frame size	Start bit	Data length	Parity/ Control bit	Stop bit
10–bit	1	7–bit	—	2
11–bit	1	7–bit	1	2

Dialog Box



SCI module
: Select either 1 or 2 (to choose module SCI1 or SCI2).

Show requested number of bytes input port
: Enables an inport (the top one if there are two) where you can input the number of bytes to request.

Maximum number of bytes
: Maximum number of bytes to receive (this is only visible if the requested number of bytes input port is enabled). This sets an upper limit on the number of bytes that will be read each time the block is called.

Show reset port
: Enables the reset input (the lower inport).

Show actual number of bytes output port
: Enables another output that shows the number of bytes actually read from the SCI buffer.

Show framing error and parity error flags
Enables another output. This output is zero if no framing or parity error occurred during the current read; it is true (1) otherwise. Note that for SCI1 only, a reset is required once a data overrun has occurred.

Show overrun flag
Enables another output. This output is true (1) if a data overrun occurred. Note that for SCI1 only, a reset is required once a data overrun has occurred.

Sample time
The time interval between samples. To inherit the sample time, set this parameter to `-1`. See "Specifying Sample Time" in the Simulink documentation for more information.

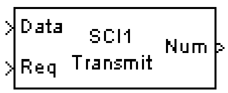
Purpose

Configure MPC555 for serial transmit, using one of QSMCM submodules SCI1 or SCI2

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Serial Communications Interface (SCI)

Description



Serial Transmit

The Serial Transmit block transmits bytes via either of the MPC555 QSMCM submodules SCI1 or SCI2. You can use it either to transmit a fixed number of bytes, or, by enabling the second input, transmit a variable number of bytes each time this block is called. With SCI1, a hardware buffer is used that allows up to 16 bytes to be queued for transmission. With SCI2, the buffer allows only up to one byte to be queued each time the block is called. Once bytes are queued for transmit, they will be sent as fast as possible by the serial interface hardware with no further intervention required by the rest of the application.

If the hardware buffer is not empty when the block is called, i.e., the previous transmission is not yet complete, then no new bytes will be queued for transmit. This condition can be identified from the "actual number of bytes" block output; if no bytes were queued for transmit, this output returns zero.

To configure the serial interface bit rate and data format, see “Serial Communications Interface (SCI) Configuration Parameters” on page 5-773.

The device driver used for the Serial Transmit block does not require the use of CPU interrupts.

Block Inputs and Outputs

The first input contains the data to be transmitted; this input signal may be either a vector or scalar with data type `uint8`. The optional second input must be a scalar and may be used to control the number of bytes transmitted. The number of bytes to transmit should not be greater than the width of the first input signal.

The block output port "actual number of bytes output" gives the number of bytes queued for transmit. If the previous transmission was complete, this number will be equal to the requested number of bytes to transmit, provided that this was less or equal to 16 in the case of SCI1, or 1 in the case of SCI2. See “Data Type Support and Scaling for Device Driver Blocks” for information on supported input/output data types and scaling of input/output signals.

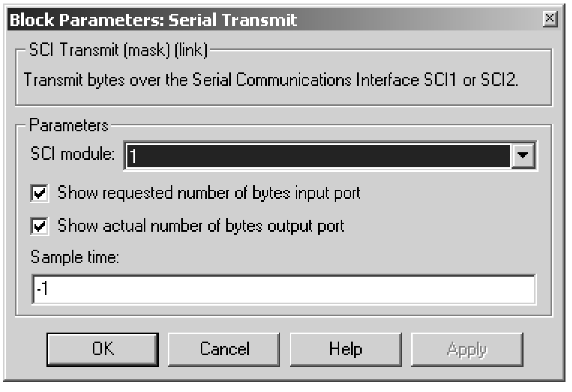
This block supports the following three serial data frame formats:

Frame size	Start bit	Data length	Parity/ Control bit	Stop bit
10–bit	1	7–bit	1	1
10–bit	1	8–bit	—	1
11–bit	1	8–bit	1	1

This block does not support the following two serial data frame formats:

Frame size	Start bit	Data length	Parity/ Control bit	Stop bit
10–bit	1	7–bit	—	2
11–bit	1	7–bit	1	2

Dialog Box



SCI module
Select either 1 or 2 (to choose module SCI1 or SCI2).

Show requested number of bytes input port
Enable/disable the input for number of bytes to send. If cleared, the number of bytes sent is just the width of the first inport; if selected, the second input is enabled, which controls the number of bytes to send.

Show number of bytes output port
Enable/disable the output port for number of bytes actually sent. If selected, this value is available from the first output.

Sample time
The time interval between samples. To inherit the sample time, leave this parameter at the default `-1`. See "Specifying Sample Time" in the Simulink documentation for more information.

MPC5xx Switch External Mode Configuration

Purpose

Configure model for external mode or executable building

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Utilities

Description

Place the Switch External Mode Configuration block in your model and double-click it to run a convenience function to configure your model for building an executable or executing your model in external mode. When you double-click the block, a dialog appears. Choose either **Building an executable** or **External mode**, and click **OK**.

When you choose building an executable, messages at the command line inform you the following steps are taken to configure your model:

1. **Inline parameters** are selected (under **Optimization > Signals and Parameters** in the Configuration Parameters dialog box). This is required for ASAP2 generation
2. **Normal** simulation mode is selected (in the Simulation menu, and drop-down list in the toolbar).
3. `ASAP2` is selected as the **Interface** (under **Code Generation > Interface**, in the Data exchange pane, in the Configuration Parameters dialog box).

When you choose external mode, messages at the command line inform you the following steps are taken to configure your model:

1. **Inline parameters** are selected (under **Optimization > Signals and Parameters** in the Configuration Parameters dialog box). This is required for external mode.
2. **External** simulation mode is selected (in the Simulation menu, and drop-down list in the toolbar).
3. `External mode` is selected as the **Interface** (under **Code Generation > Interface**, in the Data exchange pane, in the Configuration Parameters dialog box).

See “Using External Mode” for instructions for converting a model to use external mode for signal logging and parameter tuning.

MPC5xx Switch Target Configuration

Purpose

Configure model and target preferences to predefined hardware configuration

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Utilities

Description

Switch Target Hardware Configuration

Place this block in your model and double click it to run a convenience function that configures your model and Target Preferences to one of a set of predefined configurations. If your setup does not correspond to one of the predefined configurations, you may wish to use the file (`mpc555rtswitchconfig.m`) as a template for setting up your own customized configurations. The predefined configurations include settings for:

- Phytec phyCORE-MPC555 (system frequency 20 or 40 MHz)
- Phytec phyCORE-MPC565 (system frequency 40 MHz)
- Axiom CME-555 (system frequency 40 MHz)
- Axiom CME-564 (system frequency 40 MHz)

Purpose

Count transmit and receive errors detected on selected TouCAN modules

Library

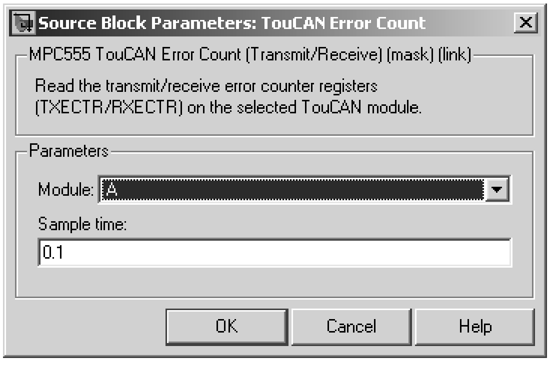
Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

Description

The TouCAN Error Count block maintains and reports a count of errors detected by the selected TouCAN module during receive and transmit. The receive and transmit error counts are output to the `RX` and `TX` outputs of the block, respectively.

The error counts also drive the TouCAN Warnings block outputs. (See MPC5xx TouCAN Warnings.)

Dialog Box



Module

Select TouCAN module `A`, `B` or `C`. Note that the MPC555 only has modules A and B. MPC56x (561-6) also have module C. An error will be thrown if you select C and your target processor does not support this.

Sample time

Sample time of the block.

MPC5xx TouCAN Fault Confinement State

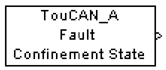
Purpose

Indicate state of TouCAN module

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

Description



TouCAN Fault Confinement State

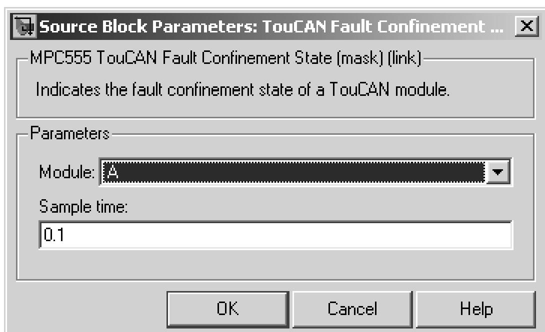
The TouCAN Fault Confinement State block provides an indicator for the state of the selected TouCAN module. The block obtains and outputs a field of two bits from the TouCAN module's Error and Status (ESTAT) register. The possible states are shown in the table below.

Refer to section 16, "CAN 2.0B Controller Module," in the *MPC555 User's Manual* for further information.

FCS State Values

State	Value	Description
Error Active	00	Normal operation
Error Passive	01	Listening only mode. The device cannot transmit.
Bus Off	1x	The device is not allowed to transmit or receive and is effectively cut off from the bus.

Dialog Box



Module

Select TouCAN module `A`, `B` or `C`. Note that the MPC555 only has modules A and B. MPC56x (561-6) also have module C. An error will be thrown if you select C and your target processor does not support this.

Sample time

Sample time of the block.

MPC5xx TouCAN Interrupt Generator

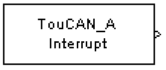
Purpose

Generate asynchronous function-call trigger when CAN interrupt occurs

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

Description



TouCAN Interrupt Generator

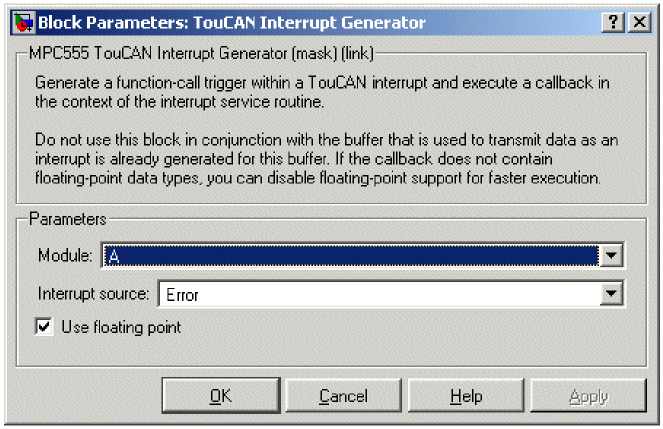
The TouCAN Interrupt Generator block generates a function-call trigger within the context of a TouCAN interrupt service routine, which can be used to asynchronously execute a function-call subsystem in the model.

This block may be used to execute a function-call subsystem on occurrence of Bus Off, Error, Wake, or buffer 0-15 interrupts.

Do not use this block unless you are aware of the dangers of using asynchronous interrupts in the model. Unpredictable data loss or model behavior may result unless extreme caution is taken. To achieve data integrity you must also place an Asynchronous Rate Transition block on each input and output of any subsystem that is triggered asynchronously by an interrupt. See MPC5xx Asynchronous Rate Transition.

For faster interrupts, you can disable floating-point support via the **Use floating point** option. However, if you disable floating-point support, do not use blocks that require floating-point operations in the function-call subsystem. Use of such blocks will cause a floating-point exception at run-time.

Dialog Box



Module

Select TouCAN module `A`, `B` or `C`. Note that the MPC555 only has modules A and B. MPC56x (561-6) also have module C. An error will be thrown if you select C and your target processor does not support this.

Interrupt source

Choose the interrupt source (Bus Off, Error, Wake or Buffer 0-15) for your ISR generator.

Use floating point

Enable or disable floating-point support.

MPC5xx TouCAN Receive

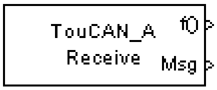
Purpose

Receive CAN messages from TouCAN module on MPC5xx

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

Description



TouCAN Receive

The TouCAN Receive block receives CAN messages from the TouCAN module.

The TouCAN Receive block can reserve any of the 16 buffers on the TouCAN module. Alternatively, you can instruct the TouCAN Receive block to select a hardware buffer automatically from the available buffers.

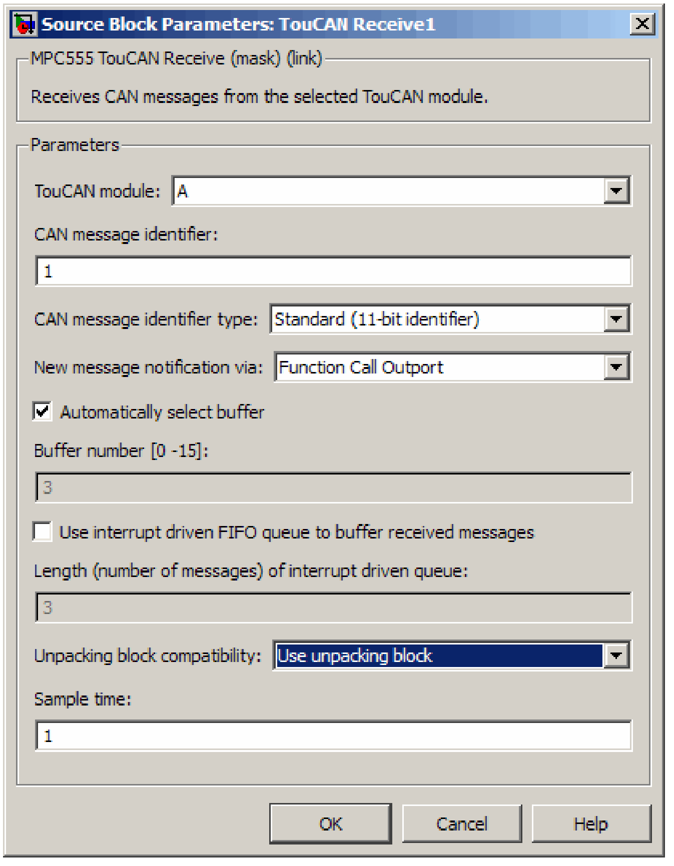
The TouCAN Receive block provides two alternative mechanisms for notifying downstream blocks that a new message has arrived. The default behavior is that the block has a Function Call Outport; in this case the associated trigger is activated whenever a new message becomes available. The alternative option is more complex and involves use of a separate TouCAN Interrupt Generator block; the TouCAN Interrupt Generator block can be used to execute the downstream function call subsystem within the context of the CAN interrupt service routine. This alternative option is recommended for advanced users only. In most applications it is recommended to use the Function Call Outport.

With the Function Call Outport mode the TouCAN Receive block polls its message buffer at a rate determined by the block's sample time. When the TouCAN Receive block detects that a message has arrived, the function call trigger is activated.

An additional option for use with the Function Call Output mode is to use a FIFO queue. In this mode, instead of polling the hardware buffer directly, the block polls a software FIFO buffer. Each time a message is received in the hardware buffer for this block an interrupt service routine automatically transfers the message to the FIFO buffer. On each block update, the FIFO is cleared by processing the messages in turn; a separate function call is generated for each message that is extracted from the FIFO. If it is known that the block sample time is smaller than the minimum time between messages that the block must receive then you should use the standard mode of operation where the hardware buffer is polled directly. However, if the messages may be arriving faster than the block is polling the buffer, you should use the FIFO mode.

Tip: if you need to receive several different messages with different identifiers, arriving at irregular intervals, into a single buffer, you can use one of the dedicated receive masks for buffers 14 or 15 along with a CAN Message Filter block, and a TouCAN Receive block operating in FIFO mode. See the Masks parameters in "TouCAN Configuration Parameters" on page 5-766.

Dialog Box



TouCAN module

Select one of the two TouCAN modules (A or B) on the MPC555. MPC56x (561-6) also have module C. The TouCAN modules can receive messages independently. Note that an error will be thrown if you select C and your target processor does not support this.

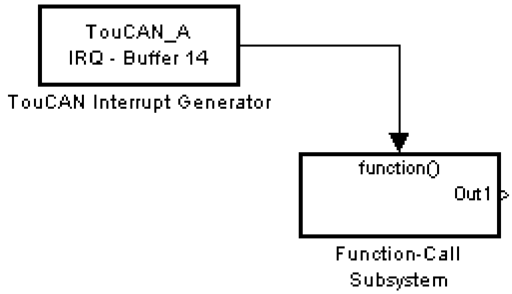
The CAN C module shares its pins with the MIOS module (which pins are shared depends on the variant). If you use the CAN C module and MIOS module together, you may experience resource conflicts which you will need to resolve.

CAN message identifier
The identifier of the message you want to receive. Note that if you have set the TouCAN configuration parameters (see "TouCAN Configuration Parameters" on page 5-766) in your model to mask out certain bits (e.g., the message identifier field) you may receive messages with identifiers other than the identifier specified here.

CAN message identifier type
The type of message you want to receive. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

New message notification via:
`Function Call Output` — Synchronous notification that a new message has arrived.

`` `TouCAN Interrupt Generator' block `` — If you select this option you must place the TouCAN Receive block in a function-call subsystem that is asynchronously triggered by a TouCAN Interrupt Generator block (as shown below). When you select this option, the function call output is no longer required, and disappears. Make sure you select the same receive buffer within the TouCAN Interrupt Generator and the TouCAN Receive block. When a message is received in the specified buffer the TouCAN Interrupt Generator block generates a function-call trigger (within the context of a TouCAN interrupt service routine), which can be used to asynchronously execute the function-call subsystem containing the TouCAN Receive block. See MPC5xx TouCAN Interrupt Generator for details.



Automatically select buffer

When this option is selected, the TouCAN Receive block automatically selects a receive buffer from the available buffers. We recommend that you use this automatic buffer selection, unless you want to use buffer 14 or 15, which can be masked individually, to receive multiple CAN message identifiers in a single buffer. See the Mask parameters in "TouCAN Configuration Parameters" on page 5-766.

Buffer number [0..15]

This field is enabled if the **Automatically select buffer** option is cleared. **Buffer number** specifies the identifier of the receive buffer for this block. We recommend that you select **Automatically select buffer** instead of manually specifying the buffer, unless you want to use buffer 14 or 15, which can be masked, to receive multiple CAN message IDs in a single buffer. See the Mask parameters in "TouCAN Configuration Parameters" on page 5-766.

Use interrupt driven FIFO queue to buffer received messages

Use the FIFO mode if the messages may be arriving faster than the block is polling the buffer. Use this option if the messages may be arriving faster than the block is polling the buffer.

Length (number of messages) of interrupt driven queue

This field is enabled if you select the interrupt driven queue option, then you can specify a number of messages.

Unpacking block compatibility

Select `Use unpacking block` or `Use message unpacking block (obsolete)`. Choose the latter only if you are using the obsolete Can Message Blocks library (`canblocks.mdl`).

Note If you have models that use CAN blocks from the obsolete Can Message Blocks library (`canblocks.mdl`), you will see an obsolescence warning message. You should update your models, as the CAN blocks may be removed in a future release

---

Sample time

Determines the rate at which to sample the buffer to see if a new message has arrived. Set to `-1` (inherited) if using this block in a function-call subsystem triggered by the TouCAN Interrupt Generator block.

Note The TouCAN Receive block sample time should be set to a value that is smaller than the minimum time between CAN messages that will be received into the corresponding buffer. If the minimum time between messages may be shorter, use the FIFO mode (select interrupt driven queue). Otherwise if more than one message is received into a buffer during a single sample interval, the older message will be overwritten.

MPC5xx TouCAN Soft Reset

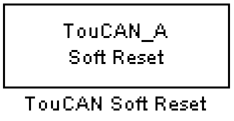
Purpose

Reset TouCAN module

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

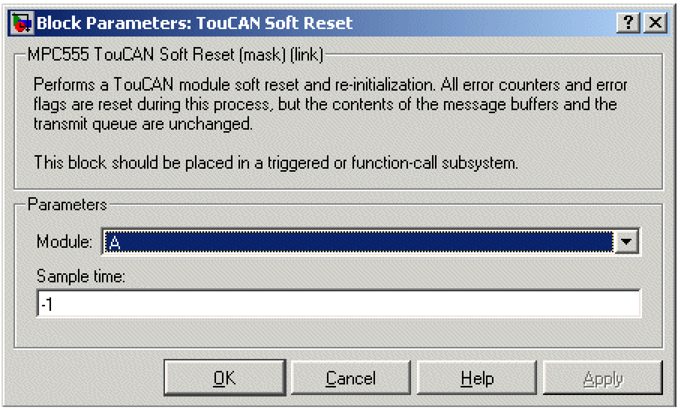
Description



When the TouCAN Soft Reset block executes, the TouCAN module resets its internal state. The TouCAN error counters will be reset. The Fault Confinement State will be reset to the Error Active state, provided the TouCAN module has not reached the Bus Off state. See MPC5xx TouCAN Fault Confinement State.

We recommend that you place this block in a triggered or function-call subsystem, with a sample time of -1 (inherited).

Dialog Box



Module

Select TouCAN module `A`, `B` or `C`. Note that the MPC555 only has modules A and B. MPC56x (561-6) also have module C. An error will be thrown if you select C and your target processor does not support this.

Sample time

Sample time of the block.

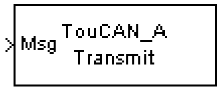
Purpose

Transmit CAN message via TouCAN module on MPC5xx

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

Description



TouCAN Transmit

The TouCAN Transmit block transmits a CAN message onto the CAN bus. The TouCAN Transmit block uses the queue set up by the MPC555 Resource Configuration object (see MPC5xx MPC555 Resource Configuration). The block should be connected to CAN Message Packing blocks. Do not ground the block or leave it unconnected. See the demos `mpc555rt_io` and `mpc555rt_candb` for an example.

The TouCAN Transmit block provides three different transmission modes. You should choose which transmission mode to use depending on the requirements of your application. The properties of each transmission mode are summarized in the following table.

Transmit Modes

	Priority Queued Transmission with Shared Buffer	Direct Transmission with Dedicated Buffer	FIFO Queued Transmission with Dedicated Buffer
Uses Interrupts	Yes	No	Yes
Configurable queue size	Yes	No	Yes

Transmit Modes (Continued)

	Priority Queued Transmission with Shared Buffer	Direct Transmission with Dedicated Buffer	FIFO Queued Transmission with Dedicated Buffer
Order of message transmission	Messages transmitted in order of priority; a new message will overwrite any existing message that is in the queue and has the same identifier and type (standard or extended)	Most recent message overwrites any unsent message in the buffer	Messages transmitted in the order that they were placed in the queue
Hardware buffers consumed	Either one or three hardware buffers are shared by many CAN Transmit blocks	One hardware buffer required for each CAN Transmit block	One hardware buffer required for each CAN Transmit block
CPU time required	Generally more than the other modes; interrupts used but time required to service interrupts is longer because it takes account of message priorities and increases with queue length	Very little; no interrupts used	Little; interrupts used but very simple interrupt service routine

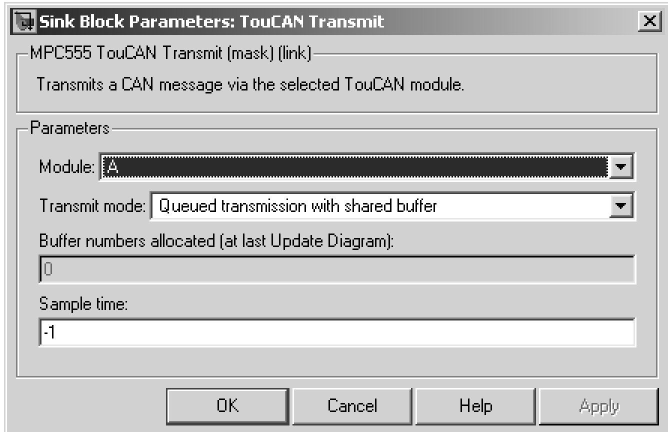
For applications where the message contains time-sensitive (e.g. real-time sensor readings) information, it is recommended to use one of the `Priority queued transmission with shared buffer` or `Direct transmission with dedicated buffer` modes. For applications where it is more important that messages are received in the order that they were queued for transmission (e.g. a data logging protocol), it is recommended to use the `FIFO queued transmission with dedicated buffer` mode.

Note that the `Queued transmission with shared buffer` mode can use one or three shared buffers depending upon the setting in the Resource Configuration block. See Transmit Shared Buffers in the TouCAN configuration settings of the MPC555 Resource Configuration object. When three buffers are used, the message entered into arbitration to be transmitted via the CAN bus is always the highest priority message available; furthermore in this mode the TouCAN module is able to transmit messages continuously by re-loading hardware buffers that become empty while another buffer is active transmitting. The shared buffer approach uses either buffer 0 or buffers 0, 1, and 2, depending on the setting in the Resource Configuration block.

If the Queued transmission with shared buffer mode is configured to use three shared buffers, there is a small possibility that some messages would be transmitted more than once. If you want to prevent this behavior, you should use this mode with a single shared buffer or use a mode other than Queued transmission with shared buffer.

The 'Queued transmission with shared buffer' mode maintains a queue of messages that are loaded into a hardware buffer of the TouCAN module as soon as one is available. Note that if a new message is ready to be sent that is higher priority than messages already in the hardware buffers then the lowest priority message will be moved from the hardware buffer back into the queue. This way, a high priority message cannot be blocked by one or more lower priority messages that are already in the hardware buffers. Under some circumstances it is possible that a lower priority message will actually be transmitted despite being moved from the hardware buffer back into the software queue; if this happens, the message concerned would be transmitted twice rather than once.

Dialog Box



Module

Select TouCAN module `A`, `B` or `C`. Note that the MPC555 only has modules A and B. MPC56x (561-6) also have module C. An error will be thrown if you select C and your target processor does not support this.

The CAN C module shares its pins with the MIOS module (which pins are shared depends on the variant). If you use the CAN C module and MIOS module together, you may experience resource conflicts which you will need to resolve.

Transmit mode

Select one of the transmit modes described in the table.

Length (number of messages) of FIFO queue

If you select the FIFO transmit mode, you can set the number of messages in the FIFO queue here. Note this is only for the FIFO queue and is not the same as the `Transmit_Queue_Length` Resource Configuration parameter in "TouCAN Configuration Parameters" on page 5-766, which only applies to shared queues.

Buffer numbers allocated (at last Update Diagram)

Read only field for information on which buffers are in use.

Sample time

Choose -1 to inherit the sample time from the driving blocks. The TouCAN Transmit block does not inherit constant sample times and runs at the base rate of the model if driven by invariant signals.

MPC5xx TouCAN Warnings

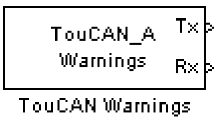
Purpose

Flag excessively high transmit or receive error counts on TouCAN modules

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ CAN 2.0B Controller Module

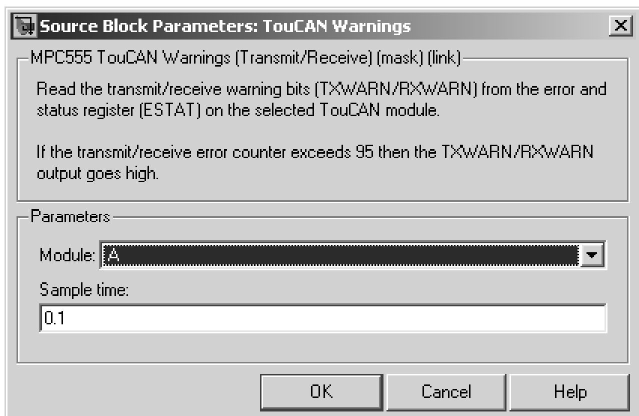
Description



The TouCAN Warnings block has two logical outputs, `RX` and `TX`. If the transmit error counter is over 95, then the `TX` output goes high. If the receive error counter is over 95, then the `RX` output goes high.

Use this block, in conjunction with a TouCAN Error Count block, to monitor error conditions on a selected TouCAN module.

Dialog Box



Module

Select TouCAN module `A`, `B` or `C`. Note that the MPC555 only has modules A and B. MPC56x (561-6) also have module C. An error will be thrown if you select C and your target processor does not support this.

Sample time

Sample time of the block.

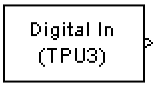
Purpose

Configure Time Processor Unit (TPU3) channel for digital input

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



TPU3 Digital In

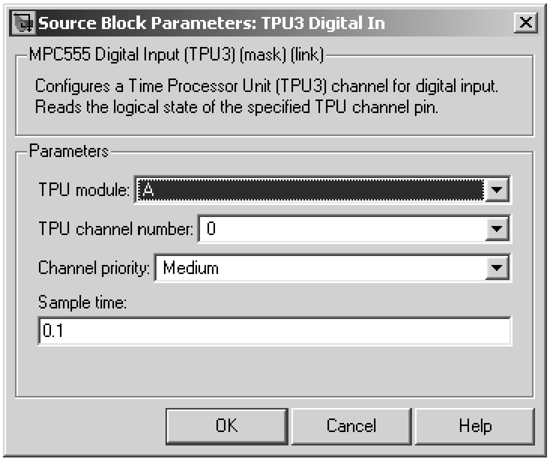
The TPU3 Digital In block reads the logical state of the selected pin (channel) on the TPU3 submodules of the MPC555 or MPC56x. You can use this block in the same way as the MIOS Digital In block. You might need to use this block instead of the MIOS Digital In block, for example, if TPU is available but not MIOS. The Channel priority field specifies a number in the range 0..15, corresponding to 16 independent timer channels on each of the modules of the TPU3. The output of the block represents the logic state of the pin referenced in the module and channels fields. When the signal on a given pin is a logical 1, the block output signal will be equal to 1; otherwise the block output element will equal zero.

The TPU has 16 channels on each module A and B (MPC565 and 566 also have module C). You can use each of these channels independently, so for an MPC555 you could use up to 32 of these blocks, specifying different channels, at once.

Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User's Manual* for further information, and the TPU3 Digital I/O Application Programming Note (search for "TPUPN18/D").

For an example showing how to use this block see the `mpc555rt_io` demo.

Dialog Box



TPU module

Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number

Choose 0-15.

Channel priority

Choose `Low`, `Medium` or `High`.

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Sample time

The default is always `0.1` for input driver blocks, but you will need to change this to suit the frequency of your input signals.

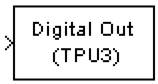
Purpose

Configure Time Processor Unit (TPU3) channel for digital output

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



TPU3 Digital Out

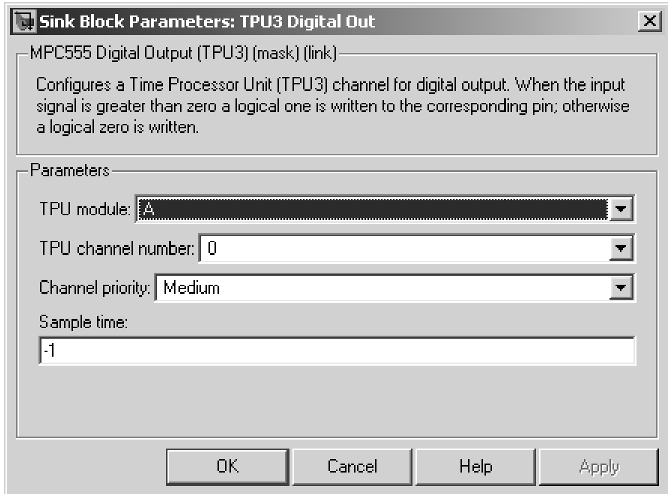
The TPU3 Digital Out block sets the state of the selected pin (channel) on the TPU3 submodule of the MPC555 (or MPC565 or MPC566). The Channel priority field specifies a number in the range 0..15, corresponding to the 16 independent channels on each TPU3 module (A, B or C). You can use each of these channels independently, so you could use up to 32 of these blocks (48 for an MPC565 or MPC566) specifying different channels at once.

When the input signal is greater than zero, a logical 1 is written to the corresponding pin. When the input signal is less than or equal to zero, a logical zero is written to the corresponding channel.

Refer to Section 17, "Time Processor Unit 3", in the *MPC555 User's Manual* and the TPU3 Digital I/O Application Programming Note (search for "TPUPN18/D") for further information about the TPU3.

For an example showing how to use this block see the `mpc555rt_io` demo.

Dialog Box



TPU Module
: Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number
: Choose 0-15.

Channel priority
: Choose `Low`, `Medium` or `High`.

  The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest first).

Sample time
: Default `-1`: this setting specifies that the block inherits its sample time from the block connected to its input (inheritance) (unless it is in a triggered subsystem). It makes no sense to sample faster

than your input is changing, so normally you should leave this at the default.

TPU Digital Out doesn't use a timebase. The output pin is written to at the rate specified by the block sample time. See "Time Processor Unit (TPU3) Configuration Parameters" on page 5-769 for details on settings for the TCR1 clock. See also the TPU3 Digital In Application Programming Note (search for "TPUPN18/D").

MPC5xx TPU3 Fast Quadrature Decode

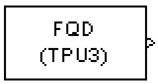
Purpose

Configure pair of TPU3 channels for Fast Quadrature Decode (FQD)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



TPU3 Fast Quadrature Decode

The TPU3 Fast Quadrature Decode block decodes position information from quadrature encoder hardware. The relative phase of a pair of input signals is used to determine direction of movement. The signals are decoded to increment or decrement the position counter (block output). You can derive a speed from the position information. It is particularly useful for decoding position and direction information from a slotted encoder in motion control systems.

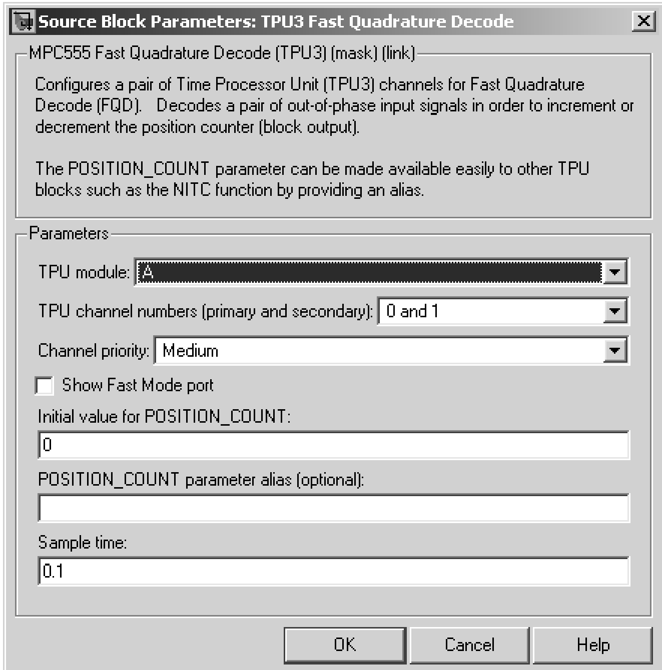
In normal mode (the default), the position counter is incremented or decremented for each valid transition on either channel. The counter increments when the primary channel is ahead and decrements when the primary channel lags. A switch in the phase relationship indicates a change of direction.

At certain speeds you may want to switch to fast mode. You can supply an input to tell the block to switch to fast mode under specified conditions. In fast mode only one of the two input signals is read. The position counter increments or decrements by 4 for each rising transition on the primary channel only (instead of once for each transition in each signal). This reduces the TPU processing load; you can also decode at more than four times the maximum count rate of normal mode.

The counter is 16 bit and free flowing (that is, it overflows to 0, and underflows to 0xFFFF). You must take care when calculating speed derived from the counter, as it may be necessary to use two's complement arithmetic. A useful document is the *TPU Fast Quadrature Decode Programming Note* — search for "*TPUPN02/D*."

It is possible to overload the TPU processor; if you observe unexpected behavior you should consult the TPU documentation. Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User's Manual* for further information.

Dialog Box



TPU module

Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel numbers (primary and secondary)

Select a pair of consecutive channels from (0 and 1) to (14 and 15). The primary channel is always the lower channel number.

Channel priority:

Choose `Low`, `Medium`, or `High`

The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Show Fast Mode port

This option is unselected by default. Left unselected, the block always operates in `Normal` mode. If you select this option, an inport appears where you can input a Boolean signal to control the mode of operation (for example, from a Stateflow subsystem): 0 or false = Normal Mode; 1 or true = Fast Mode.

Fast mode conserves TPU activity by only reading one of the two signals. This also allows you to decode at more than four times the maximum count rate of `Normal` mode. This may be appropriate at some speeds where you can assume the behavior of the second sign — instantaneous direction change is assumed to be impossible. The counter is updated in the same direction as when the last transition was serviced in Normal Mode. The position counter is incremented or decremented by 4 for every rising transition read on the primary channel, instead of having to read all four transitions in the two signals.

Initial value for POSITION_COUNT

Set an initial value. Range checking is applied (must be 16 bit).

POSITION_COUNT parameter alias (optional)

Provide a name that blocks such as the TPU3 New Input Capture/Input Transition Counter can use to refer to the `POSITION_COUNT` Fast Quadrature Decode parameter (see MPC5xx TPU3 New Input Capture/Input Transition Counter). Using a name is clearer than using absolute channel and parameter indices to refer to the position count from another TPU block.

Sample time

The default is always `0.1` for input driver blocks, but you will need to change this to suit the frequency of your input signals.

This block uses TCR1 as a timebase, but the functionality of the TPU Fast Quadrature Decode (FQD) function used by the block is not changed by changing the speed of the TCR1 clock. The Position Count output is incremented at a rate entirely controlled by the rising and falling edges of the pair of input waveforms (and the Fast mode input). See “Time Processor Unit (TPU3) Configuration Parameters” on page 5-769 for more information on the TCR1 timebase settings.

MPC5xx TPU3 New Input Capture/Input Transition Counter

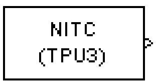
Purpose

Configure Time Processor Unit (TPU3) channel for New Input Capture/Input Transition Counter (NITC)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



TPU3 New Input Capture/ Input Transition Counter

The TPU3 New Input Capture/Input Transition Counter block counts transitions on the input pin and/or captures a TCR timebase value or a TPU parameter RAM value after a certain number of transitions. You can select the number of transitions and whether to capture on rising or falling transitions or both.

You can select up to three outputs to display. Each will have a separate outport:

- `FINAL_TRANS_TIME` shows the captured value each time the maximum number of transitions (`MAX_COUNT`) is reached
- `TRANS_COUNT` shows the number of transitions counted (resets each time `MAX_COUNT` is reached)
- `LAST_TRANS_TIME` shows the captured value at the most recent transition, updated at every transition (except final transitions). At the final transition `LAST_TRANS_TIME` shows the captured value at the previous transition.

You can choose whether to capture the TCR1 timebase value each time the `MAX_COUNT` number of transitions is reached, or you can specify the address of a TPU parameter in RAM to capture at that moment. Note this block always operates in continuous mode, not single-shot — transitions are counted up to `MAX_COUNT` and then the block resets and continues counting from zero.

It is possible that the three outputs are not read coherently. They are read one after another, and it is possible that while the memory is accessed for one parameter the next to be read may have changed value. This depends on the speed of your input signal. This should not be important for most purposes because only `TRANS_COUNT` or `FINAL_TRANS_TIME` will be the outputs of interest.

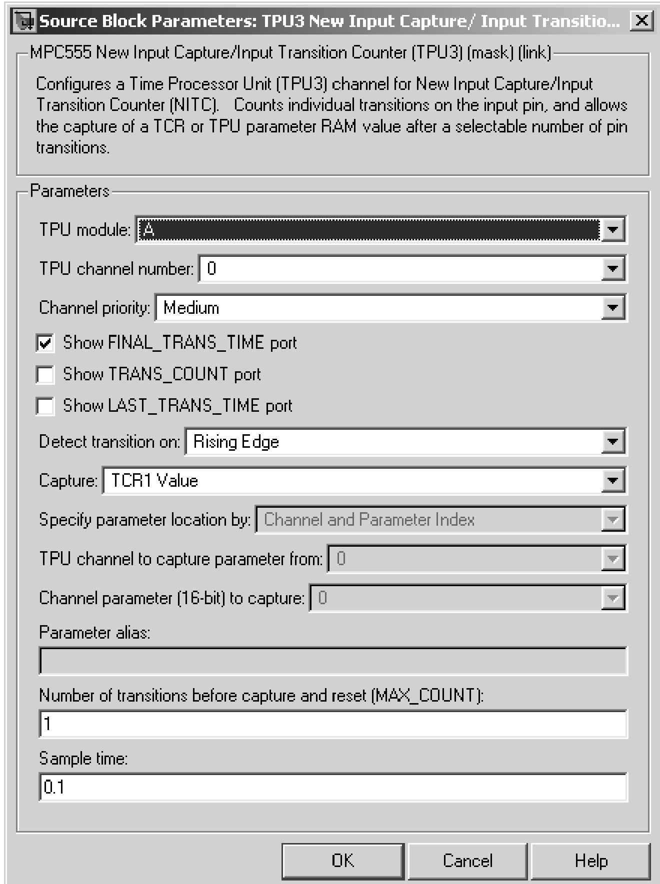
As an example, you could use this block in conjunction with the TPU3 Fast Quadrature Decode block for calibration purposes. Quadrature encoders often generate an index signal in addition to the pair of signals whose relative phase contains the position information. You could put this index signal into an NITC input to count pulses in order to calibrate the position of the encoder.

Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User's Manual* for further information. A particularly useful document is the *TPU New Input Capture/Input Transition Capture Programming Note* — search for "*TPUPN08/D*." Look in the appropriate TPU programming note to look up parameter addresses if you want to capture TPU Parameters instead of TCR1 clock ticks.

As an example of using TPU parameters, if you wanted to use this block to capture the position count from a TPU Fast Quadrature Decode block, you need to set the correct channel number and parameter address. You must set the channel number to the primary FQD channel (FQD blocks use a pair of channels, the first is primary). Each TPU channel can have up to eight parameters (0 through 7), in this case you must choose parameter 1 (`POSITION_COUNT`).

MPC5xx TPU3 New Input Capture/Input Transition Counter

Dialog Box



TPU module

Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number

Choose 0-15.

Channel priority:

Choose Low, Medium, or High

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Show FINAL_TRANS_TIME port

Outputs the value captured each time the maximum number of transitions (MAX_COUNT) is reached. This value is only captured when MAX_COUNT is reached.

Show TRANS_COUNT port

Outputs the number of transitions counted. Resets to zero each time MAX_COUNT is reached.

Show LAST_TRANS_TIME port

Outputs the captured value at the latest transition. This is updated at every transition except the final one.

Detect transition on:

Choose from Rising Edge, Falling Edge or Either Edge.

Capture:

TCR1 Value — captures the value of the TCR1 timebase. See "Time Processor Unit (TPU3) Configuration Parameters" on page 5-769 for information on setting the TCR1 timebase.

Parameter RAM Value — captures the value of a TPU parameter in RAM. If you select this option you enable the parameters to choose the TPU channel number and parameter address, or to specify a parameter alias.

Specify parameter location by

Channel and Parameter Index — if you select this option you enable the two parameters to specify which TPU channel (from 0-15) and which parameter index (out of up to eight parameters per TPU channel) you want.

`Parameter Alias` — If you select this option you enable the **Parameter alias** edit box. For example you can specify a parameter alias for the POSITION_COUNT parameter in the TPU3 Fast Quadrature Decode block. See MPC5xx TPU3 Fast Quadrature Decode.

Note that you cannot set the parameter location unless you have chosen `Parameter RAM Value` for the **Capture** parameter.

TPU channel to capture parameter from
Specify which TPU channel (from 0-15) you want. This option is enabled when you choose to specify parameter location by `Channel and Parameter Index`.

Channel parameter (16-bit) to capture
Specify which parameter index (out of up to eight parameters per TPU channel) you want. This option is enabled when you choose to specify parameter location by `Channel and Parameter Index`.

Parameter alias
This option is enabled when you choose to specify parameter location by `Parameter Alias`. Enter the required alias in the edit box. For example you can specify a parameter alias for the POSITION_COUNT parameter in the TPU3 Fast Quadrature Decode block. See MPC5xx TPU3 Fast Quadrature Decode.

Number of transitions before capture and reset (MAX_COUNT)
This must be a 16-bit number specifying how many transitions to count before capturing and then resetting. A zero will be equivalent to 1 (you cannot count zero transitions) and you must not exceed the maximum of a uint16 number. The range of an unsigned 16-bit number is 0-65535 (because 65535 = (2^16) - 1).

Range checking is applied; you will receive a warning if you input an unsuitable number.

Sample time

Be sure to set the sample time fast enough not to miss any transitions. This will depend on the frequency of your input signal.

MPC5xx TPU3 Programmable Time Accumulator

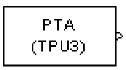
Purpose

Configure Time Processor Unit (TPU3) channel for Programmable Time Accumulator (PTA)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



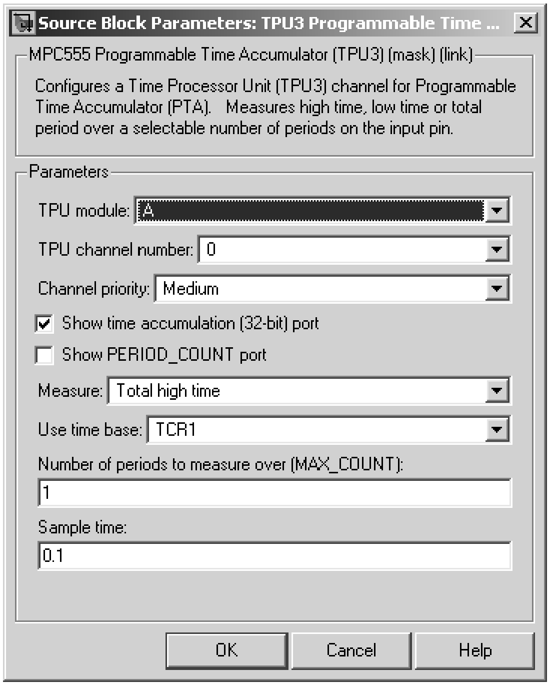
TPU3 Programmable Time Accumulator

The TPU3 Programmable Time Accumulator block reads an input pin and measures an accumulation of time over a specified number of periods - either high time, low time, or the total time. You can output the accumulated time, the number of periods, or both. You can choose whether to start counting total period on a rising or falling edge.

The accumulated time value will be read at most once between any two model steps. TPU interrupts are used so that the 32-bit output is updated only when an accumulation is complete. Therefore, the values of the parameters `HW` and `LW`, combined to create the 32-bit output, are coherent. This block is under MPC555 Resource Configuration object control, and you will receive a warning if you have not enabled TPU interrupts. If your model contains any PTA blocks, you must change the TPU IRQ settings to enable interrupts. See “Time Processor Unit (TPU3) Configuration Parameters” on page 5-769.

Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User’s Manual* for further information. A particularly useful document is the *Programmable Time Accumulator TPU Function (PTA) Programming Note* — search for "TPUPN06/D."

Dialog Box



TPU module

Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number

Choose 0-15

Channel priority:

Choose `Low`, `Medium`, or `High`

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Show time accumulation (32-bit) port
Outputs the 32-bit time accumulation value (in TCR1 clock ticks) each time `MAX_COUNT` is reached. Whether the accumulation measures high time, low time or total time depends on the **Measure** setting.

Show PERIOD_COUNT port
Outputs the number of periods counted.

Measure:
Choose from `Total high time`, `Total low time`, `Total period (starting on rising edge)`, `Total period (starting on falling edge)`.

Use time base
Select `TCR1` or `TCR2`. You can configure `TCR2` to use an external clock. See "Time Processor Unit (TPU3) Configuration Parameters" on page 5-769.

Number of periods to measure over (MAX_COUNT):
Set the number of periods to accumulate time over, up to a maximum of 255. The value is read each time `MAX_COUNT` is reached. Note that `MAX_COUNT` is 8-bit here (it is 16-bit in the TPU3 New Input Capture/Input Transition Counter block).

Sample time:
Make sure you set a sample time fast enough not to miss any periods, depending on the frequency of your input signal.

MPC5xx TPU3 Pulse Width Modulation Out

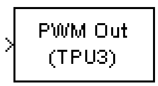
Purpose

Configure Time Processor Unit (TPU3) channel for pulse width modulation (PWM) output

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



TPU3 Pulse Width Modulation Out

The TPU3 Pulse Width Modulation Out block is used for Pulse Width Modulation (PWM) output from the TPU3 modules. You can use this block in the same way as the MIOS PWM Out block, and with the TPU block you can also vary the period dynamically using a block inport. You can modulate up to 16 of these for each module (A, B or C) using any of the independent TPU channels.

A PWM signal is a rectangular waveform whose period may or may not be constant, and whose duty cycle can be varied, under control of a modulator signal, between 0% and 100%. You can either control the period register directly, or enter the desired (ideal) period and the mask will solve for the best values for the period register. Note for the MIOS Pulse Width Modulation Out block the period is constant, but with the TPU Pulse Width Modulation Out block you can also vary the period of the PWM signal (using the `input port for pulse period` option you can supply the period as an input).

The TPU3 Pulse Width Modulation Out block acts as the modulator, controlling the duty cycle and period of the signal on the output channel. There can be one or two inputs. Input one (top) is always the duty cycle. Here an input signal in the range 0 to 1 generates a PWM output with corresponding duty cycle. Input signals outside this range cause the duty cycle to saturate at 0% or 100%.

You can specify the period register manually in the mask. If you select the option `use input port for pulse period register value`, input two appears. Here you can supply the period as an input, instead of specifying the period in the mask. PWMPER input (either block input or specified as a mask variable) must be 16 bit values in the range 0 <= PWM Period Register Value <= 32768 (0x8000).

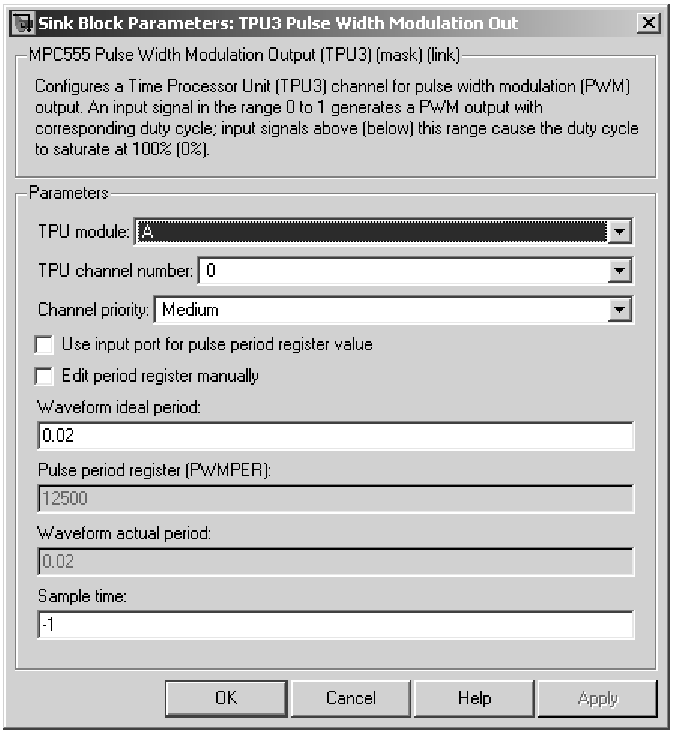
This saturation means that the block will not allow you to enter a value for PWMPER > 0x8000, or a value for `ideal period` that makes the PWMPER register go outside this range.

The TPU Pulse Width Modulation Out block uses TCR1 as a timebase for creating the output waveform. By changing the speed of the TCR1 clock, the range of available PWM periods changes. See “Time Processor Unit (TPU3) Configuration Parameters” on page 5-769 for more information on settings for the TCR1 clock.

Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User’s Manual* for further information. See also the relevant TPU3 Application Programming Note (search for "TPUPN17/D").

For an example showing both ways to use this block (specifying the period, and using the PWMPER port to input the period), see the `mpc555rt_io` demo.

Dialog Box



TPU Module

Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number

Choose 0-15

Channel priority

Choose `Low`, `Medium`, or `High`

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Use input port for pulse period register value

If you select this box, the parameters relating to setting the period register disappear because they are no longer used.

A new inport appears on the block when you select this option. Here you can input the period register value. Saturation is applied: 0 <= x <= 32768 (0x8000). You can see an example of the block in the demo model `mpc555rt_io`.

Edit period register manually

If you select this check box, you can set the **Pulse period register** parameter.

Waveform ideal period

The default is `0.02`. You can enter the waveform period you want by typing in this edit box. From this the period register is calculated and appears in the **Pulse period register (PWMPER)** edit box. The actual waveform period is also calculated and displayed, see below.

Pulse period register (PWMPER)

The default is `12500`. You can enter a value for the period register here (0<= x <= 32768 (0x8000)) only if you select **Edit period register manually**. The actual waveform period is calculated and displayed in the actual period field. If **Edit period register manually** is not selected, this edit box is disabled (gray).

Waveform actual period

You can never enter anything in this box (so it is always gray) — it is there purely to inform you, and does not affect the model code. You might find this information useful because actual and ideal waveform period are not always the same — the ideal period you enter may not always be possible.

Sample time

The default is -1: This setting specifies that the block inherits its sample time from the block connected to its input (inheritance) (unless it is in a triggered subsystem). It makes no sense to sample faster than your input is changing, so normally you leave this at the default.

MPC5xx TPU3 Rectangular Wave

Purpose

Configure Time Processor Unit (TPU3) channel for Rectangular Wave Output (RECTW)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description

TPU3 Rectangular Wave

This block is provided as an example along with the demo model mpc555rt_tpu_emu. The rectangular wave function is not part of the standard ROM mask of TPU functions but can be downloaded to DPTRAM and used by the TPU in emulation mode.

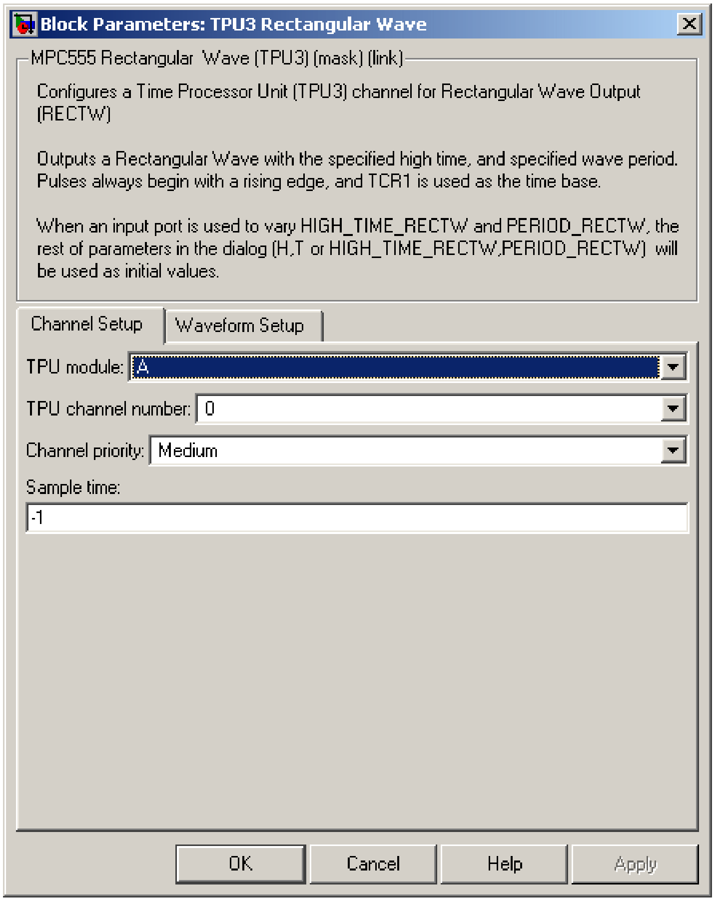
The TPU3 Rectangular Wave block outputs a rectangular wave with a specified high time and specified wave period. Pulses always begin with a rising edge, and TCR1 is used as the timebase. You can either control the high-time and waveform period registers directly, or enter the desired (ideal) periods and the mask will solve for the best values for the period registers.

If you select the option **Use input port to vary HIGH_TIME_RECTW and PERIOD_RECTW**, two inputs appear. You can use these to vary the high-time and waveform period. The rest of the parameters in the mask are used as initial values. Input 1 (top) is the high time and input 2 is the period. Inputs must be 16 bit values in the range 0 <= x <= 32768 (0x8000).

The TPU Rectangular Wave block uses TCR1 as a timebase for creating the output waveform. By changing the speed of the TCR1 clock, the range of available waveform periods changes. See “Time Processor Unit (TPU3) Configuration Parameters” on page 5-769 for more information on settings for the TCR1 clock.

Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User’s Manual* for further information.

Dialog Box



On the **Channel Setup** tab:

TPU Module

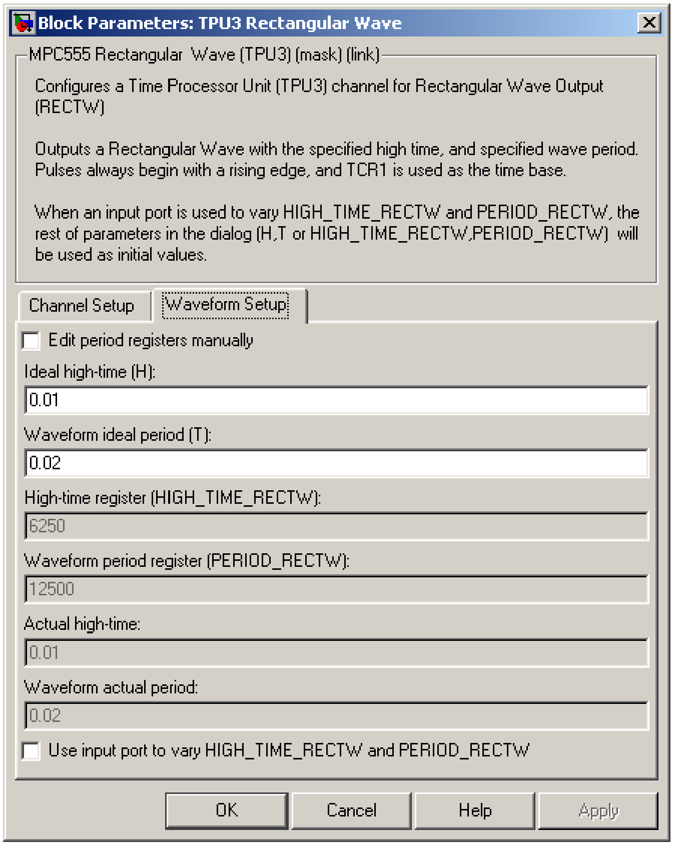
Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number
Choose 0-15

Channel priority
Choose `Low`, `Medium`, or `High`

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Sample time
The default is -1. This setting specifies that the block inherits its sample time from the block connected to its input (inheritance) (unless it is in a triggered subsystem). It makes no sense to sample faster than your input is changing, so normally you leave this at the default.



On the **Waveform Setup** tab:

Edit period registers manually

If you select this check box, you can manually set the **High-time register** and **Waveform period register** parameters.

Ideal high-time (H)

You can enter an ideal high-time period (in seconds). From this the high-time register is calculated and appears in the **High-time**

register (HIGH_TIME_RECTW) edit box. The actual waveform period is also calculated and displayed, see below.

Waveform ideal period (T)
Enter the waveform period you want by typing in this edit box. From this the waveform period register is calculated and appears in the **Waveform period register (PERIOD_RECTW)** edit box. The actual waveform period is also calculated and displayed, see below.

High-time register (HIGH_TIME_RECTW)
You can enter a value for the high-time register here (0<= x <= 32768 (0x8000)) only if you select **Edit period registers manually**. The actual high-time period is calculated and displayed in the actual high-time period field.

Waveform period register (PERIOD_RECTW)
You can enter a value for the period register here (0<= x <= 32768 (0x8000)) only if you select **Edit period registers manually**. The actual waveform period is calculated and displayed in the actual period field.

Actual high-time
Information field. You might find this information useful because actual and ideal high-time period are not always the same — the ideal period you enter may not always be possible.

Waveform actual period
Information field. You might find this information useful because actual and ideal waveform period are not always the same — the ideal period you enter may not always be possible.

Use input port to vary HIGH_TIME_RECTW and PERIOD_RECTW
Select this box to use input ports to control the high-time and waveform period registers. Two input ports appear on the block (the top input is high-time).

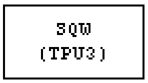
Purpose

Configure Time Processor Unit (TPU3) channel for Square Wave Output (SQW)

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx/ Time Processor Unit (TPU3)

Description



TPU3 Square Wave

This block is provided as an example along with the demo model `mpc555rt_tpu_emu`. The square wave function is not part of the standard ROM mask of TPU functions but can be downloaded to DPTRAM and used by the TPU in emulation mode.

The TPU3 Square Wave block outputs a square wave with a specified high time (and corresponding low time). Pulses always begin with a rising edge, and TCR1 is used as the timebase.

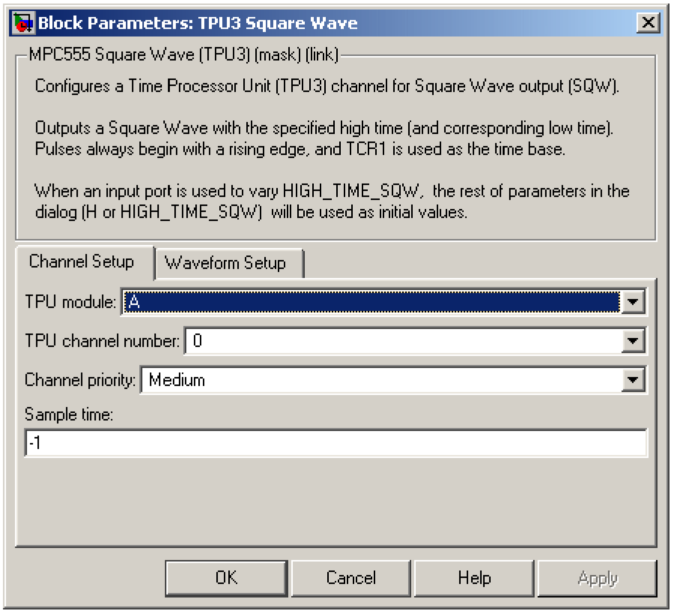
You can either control the high-time register directly, or enter the desired (ideal) period and the mask will solve for the best values for the period register.

If you select the option **Use input port to vary HIGH_TIME_SQW**, an input appears. You can use this input to vary the high-time. The rest of the parameters in the mask are used as initial values. The input must be a 16 bit value in the range 0 <= x <= 32768 (0x8000).

The TPU Square Wave block uses TCR1 as a timebase for creating the output waveform. By changing the speed of the TCR1 clock, the range of available waveform periods changes. See "Time Processor Unit (TPU3) Configuration Parameters" on page 5-769 for more information on settings for the TCR1 clock.

Refer to Section 17, "Time Processor Unit 3," in the *MPC555 User's Manual* for further information.

Dialog Box



On the **Channel Setup** tab:

TPU Module

Select TPU module A, B or C; each has 16 channels. Note that the MPC555 only has modules A and B. MPC565 and MPC566 also have module C. An error will be thrown if you select C and your target processor does not support this.

TPU channel number

Choose 0-15

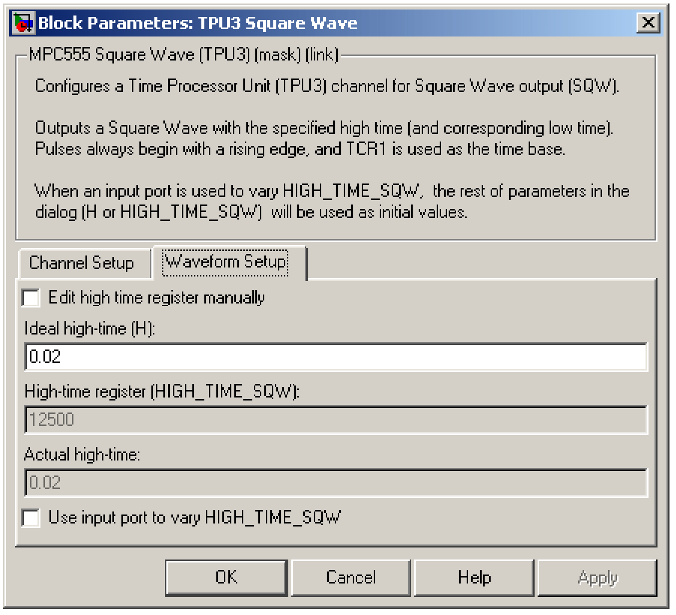
Channel priority

Choose `Low`, `Medium`, or `High`

The host CPU makes a channel active by assigning it one of the three priorities. You choose the order in which channels are serviced by setting the channel number and assigned priority. The order in which channels are serviced is determined by assigned priority first, followed by channel number (lowest number first).

Sample time

The default is -1. This setting specifies that the block inherits its sample time from the block connected to its input (inheritance) (unless it is in a triggered subsystem). It makes no sense to sample faster than your input is changing, so normally you leave this at the default.



On the **Waveform Setup** tab:

Edit high-time register manually

If you select this check box, you can manually set the High-time register (HIGH_TIME_SQW) parameter.

Ideal high-time (H)

You can enter an ideal high-time period (in seconds). From this the high-time register is calculated and appears in the High-time register (HIGH_TIME_SQW) edit box. The actual waveform frequency is also calculated and displayed, see below.

High-time register (HIGH_TIME_SQW)

You can enter a value for the high-time register here (0<= x <= 32768 (0x8000)) only if you select **Edit high-time register manually**. The actual high-time period is calculated and displayed in the actual high-time field.

Actual high-time

Information field. You might find this information useful because actual and ideal high-time period are not always the same — the ideal period you enter may not always be possible.

Use input port to vary HIGH_TIME_SQW

Select this box to use an input port to control the high-time register. An input port appears on the block.

MPC5xx Watchdog

Purpose

In case of application failure, time out and reset processor

Library

Embedded Coder/ Embedded Targets/ Processors/ Freescale MPC5xx

Description

Watchdog

The Watchdog block lets you set the time-out period for the watchdog timer. The watchdog timer is a safety feature that is used to monitor correct behavior of the application. The timer is loaded with an initial value and counts down from this value. If the timer ever reaches zero, a watchdog time-out occurs, forcing a processor reset.

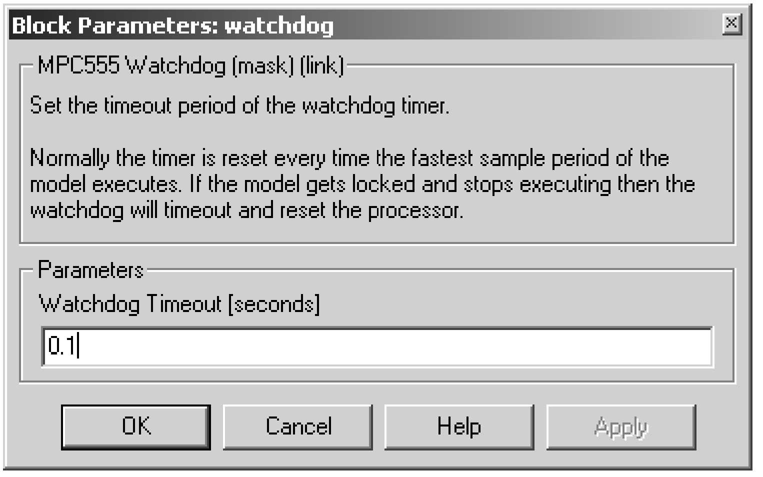
In normal operation, the watchdog timer is serviced at a regular interval (each model step) by the application code; this occurs at a higher frequency than the **Watchdog Timeout** parameter period. Therefore the counter never reaches zero and a processor reset is never triggered.

In the event of a software failure that causes the application to lock up, the watchdog timer will not be serviced. Therefore, it will time out when the counter reaches zero. This in turn causes a processor reset, which restarts the application.

You do not need to include a Watchdog block in your model unless you want to change the **Watchdog Timeout** parameter period to a value other than the default. By default, the watchdog timer is enabled and the time-out period is set to the largest possible value, which is several seconds, depending on system frequency.

Note that the Watchdog block has neither input nor output connections.

Dialog Box



Watchdog Timeout

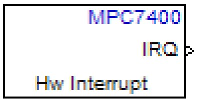
The **Watchdog Timeout** period must be set to a value that is larger than the fastest sample rate in the system, because this is the rate at which the watchdog timer is serviced. To set the **Watchdog Timeout** period, place a Watchdog block anywhere in the model and open its dialog box.

Purpose

Generate Interrupt Service Routine

Library

Block Library: idelinklib_ghsmulti



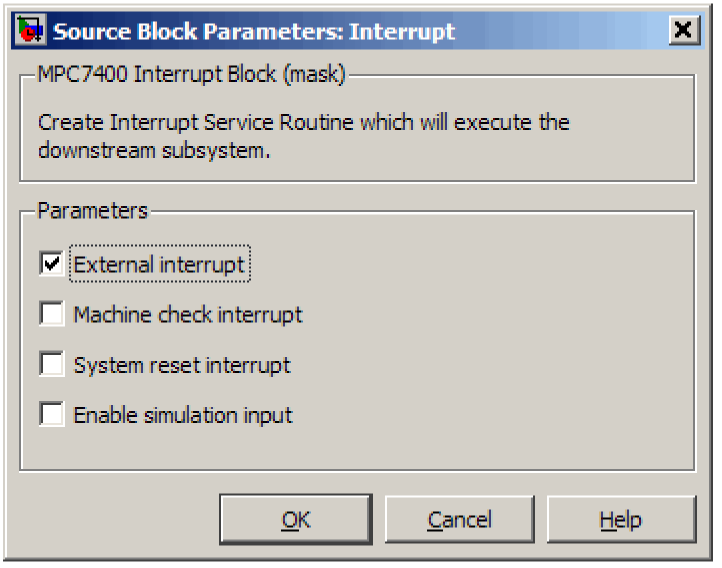
Interrupt

Description

The block creates ISRs for three processor interrupts—External, Machine check and System reset. When you incorporate this block in your model, code generation results in ISRs on the processor that run the blocks downstream from this block. For more information about these interrupts, refer to your MPC7400 documentation.

When you enable more than one interrupt on the block dialog box, the block multiplexes the ISR outputs onto the output port on the block. To resolve the different ISRs, connect the output port IRQ to a Demux block. Connect the demultiplexed outputs to downstream blocks or subsystems. Refer to Examples to see the multiple interrupt configuration in a model.

Dialog Box



External interrupt
: Interrupt generated by an external system that asserts the `intr` pin of the 7400 microprocessor.

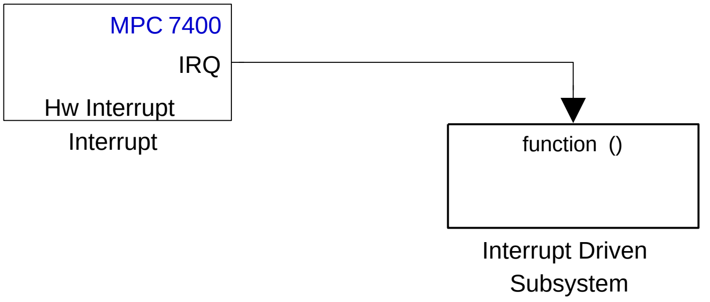
Machine check interrupt
: Enable the asynchronous, nonmaskable machine check exception provided by the processor. The exception responds to the conditions described in the MPC7400 documentation.

System reset interrupt
: Enable the asynchronous, nonmaskable System interrupt exception provided by the processor. The exception responds to the conditions described in the MPC7400 documentation.

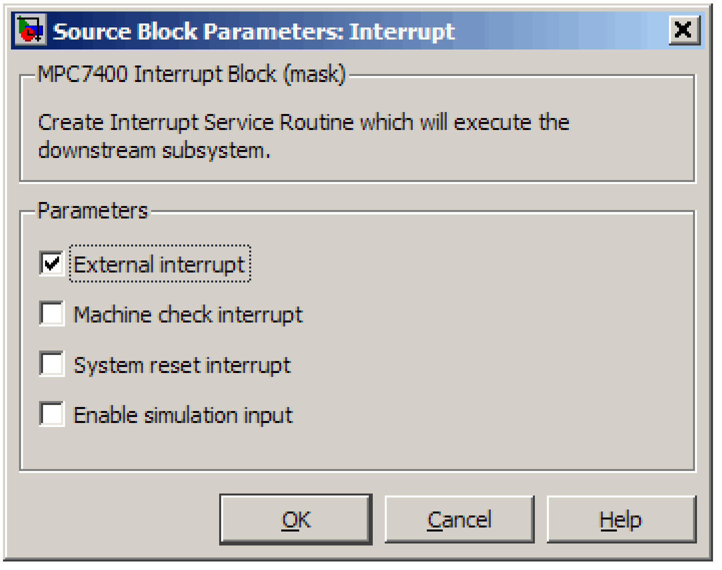
Enable simulation input
: Select this option to have Simulink add an input port to the HW Interrupt block. This port receives input only during simulation. Connect one or more simulated interrupt sources to the input to drive the model interrupt processing.

Example

The following model shows the HW Interrupt block triggering a subsystem. The interrupt block is configured to respond to external interrupts.

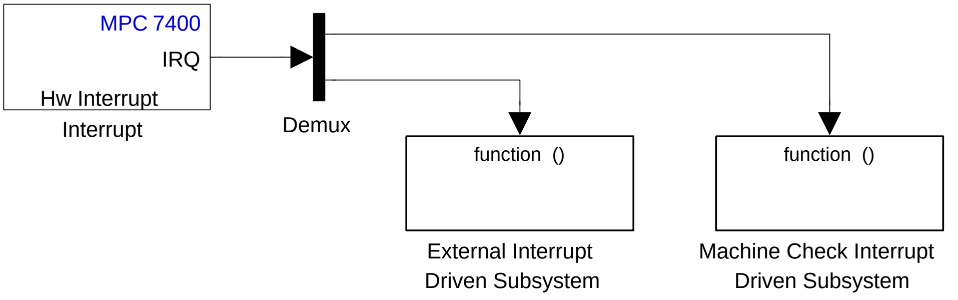


Here is the block mask.

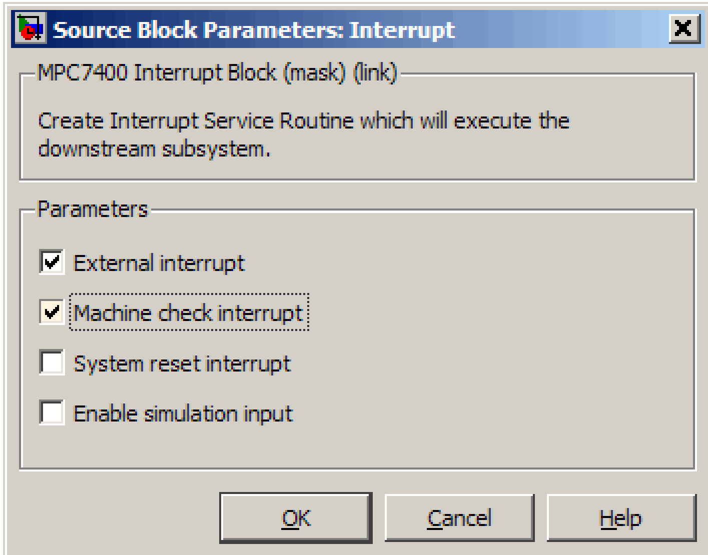


When your peripherals assert the external interrupt pin on the processor, the code generated by the HW Interrupt block during the project build process accepts the interrupt and triggers the attached subsystem through an ISR.

When you select more than one interrupt, connect the output of the block to a Demux block to separate the ISRs, as shown in the following model:



Here is the block mask showing the external and Machine check interrupts selected.



To test your interrupt configuration in simulation, select **Enable simulation input** on the block dialog box and then input a signal to the block to simulate the external interrupt.

See Also

Idle Task, Memory Allocate, Memory Copy

Mode Switch for Invoke AUTOSAR Server Operation

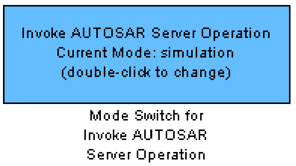
Purpose

Toggle AUTOSAR client-server operation subsystem blocks between simulation and code generation mode

Library

Embedded Coder/ AUTOSAR

Description



You can add this switch block to your Simulink model that contains client-server subsystem blocks. Double-click the switch block to toggle client-server blocks between simulation and code-generation mode.

Parameters

Configure the model for

Value selected from

- `code generation`
- `simulation`

For this block, `code generation` is selected by default.

See Also

Invoke AUTOSAR Server Operation

"Configuring Client-Server Communication" in the Embedded Coder documentation

Purpose

Generate Interrupt Service Routine

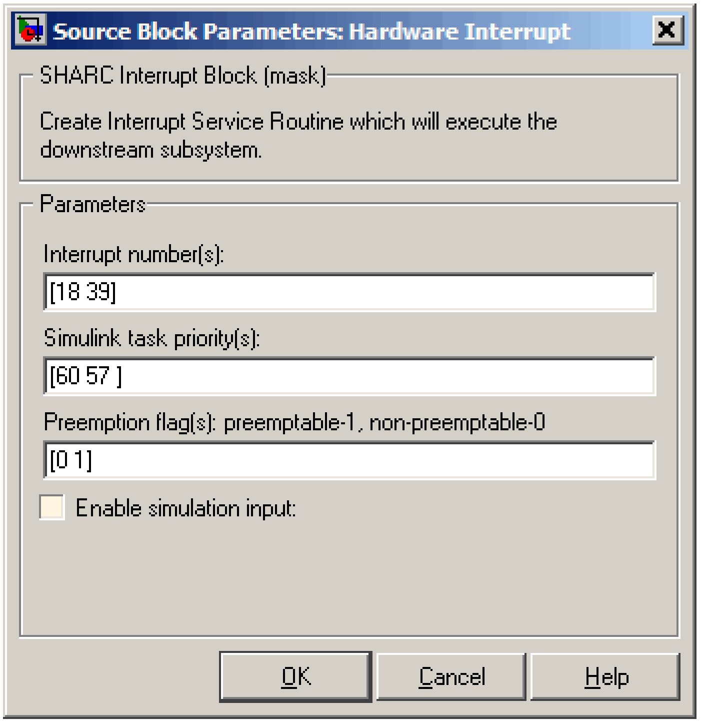
Library

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**.

Proper code generation requires rate transition code (refer to Rate Transitions and Asynchronous Blocks in the Simulink Coder documentation). The task priority values ensure 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

Purpose

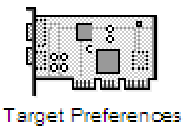
Configure model for specific IDE, tool chain, board, and processor

Library

Simulink Coder/ Desktop Targets

Embedded Coder/ Embedded Targets

Description



Use the Target Preferences block to configure a model to for a specific IDE/tool chain, board, and processor. Your MathWorks software depends on this information to properly simulate the model and generate code for your environment.

The appearance and contents of the Target Preferences block varies widely, depending on the options you have selected. The following sections describe all of the user interface elements in the Target Preferences block, even though the Target Preferences block cannot simultaneously display all of the user interface elements.

For more information, see the Target Preferences topic in the User's Guide.

Note The following actions update the appropriate model Configuration Parameters with new values:

- Adding a Target Preferences block to your model and clicking Yes in the **Initialize Configuration Parameters** dialog box.
- Opening the Target Preferences block in your model and selecting a new **IDE/Tool Chain**.
- Opening the Target Preferences block in your model and applying changes to the **Board** and **Processor** parameters.

Note If you are using a Windows host, use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Note The figures in this documentation include references to various third-party vendors and products. These images aid with recognition of specific user interface elements. Do not infer a preference or endorsement for any vendor or product over another.

Dialog Boxes

This reference page section contains the following subsections:

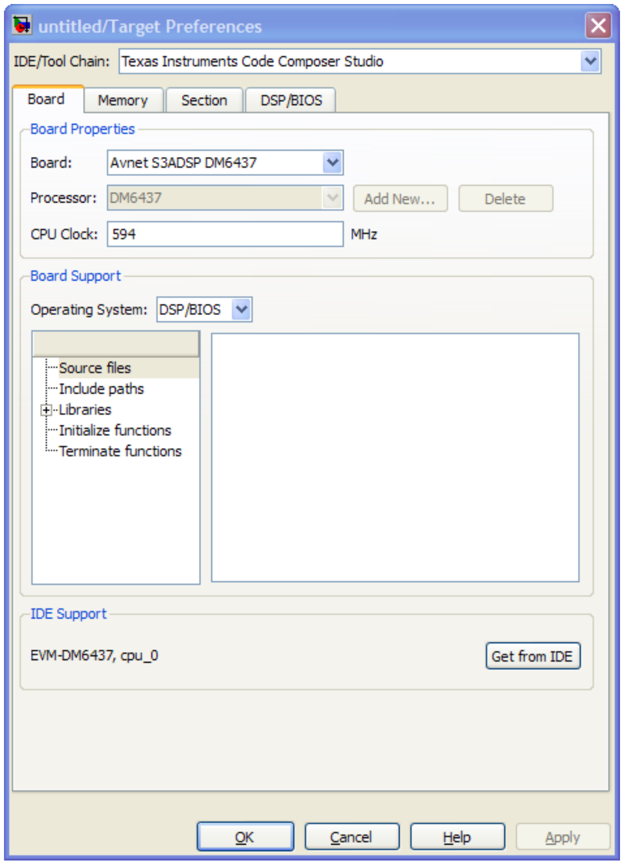
Use the **IDE/Tool Chain** parameter to select the Integrated Development Environment (IDE) or software build tool chain with which you are working. Selecting any option automatically applies that selection to the Target Preferences block and updates the panes and options the block displays.

Target Preferences block dialog box provides tabbed access to the following panes:

- Board Pane — Select the target board, processor, clock speed, and, in some cases, RTOS. In addition, **Add new** on this pane opens the **Add Processor** dialog box.
- Memory Pane — Set the memory allocation and layout on the processor (memory mapping).
- Section Pane — Determine the arrangement and location of the sections on the processor and compiler information.
- DSP/BIOS Pane — For Texas Instruments CCS IDE and C6000 processors: Specify how to configure tasking features of DSP/BIOS™.
- Peripherals Pane — For Texas Instruments CCS IDE and C2000 processors: Specify how to configure the peripherals provided by C2xxx processors, such as the SPI_A, SPI_B, GPIO, or eCAP peripherals.
- Linux Pane — For the Eclipse IDE: Specify the scheduling mode and base rate task priority of the software to run on a Linux target.

- VxWorks Pane — For the Wind River Diab/GCC (makefile generation only): Specify the scheduling mode of the software to run on a VxWorks target.

Board Pane



The following options appear on the **Board** pane, which has separate panels for **Board Properties**, **Board Support**, and **IDE Support** labels.

Board

Select your target board from the list of options. Selecting a specific board sets the appropriate value for the **Processor** parameter. If you select a custom board, also set the **Processor** parameter to an appropriate value.

Processor

The Board and Processor settings apply default values to many of the remaining Target Preferences parameters, such as those under the **Memory** and **Section** tabs.

If the coder product supports an operating system for the processor, it enables the **Operating system** option.

If you are using the Eclipse IDE and set **Processor** to `Generic/Custom`, open the model Configuration Parameters and use the **Hardware Implementation** pane to define the custom hardware. With this approach, hardware support depends on the Simulink Coder product, not on the coder product. For more information, see "Hardware Implementation Pane".

Note Selecting or reselecting a processor resets the solver and some processor-specific parameters to their default values.

Add New

Clicking **Add new** opens a new dialog box where you specify configuration information for a processor that is not on the Processor list.

For details about the New Processor dialog box, refer to "Add Processor Dialog Box" on page 5-941.

Delete

Delete a processor that you added to the **Processor** list. You cannot delete any of the standard processors.

CPU Clock

Enter the actual clock rate the board uses. This action does not change the rate on the board. Rather, the code generation process requires this information to produce code that runs correctly on the hardware. Setting this value incorrectly causes timing and profiling errors when you run the code on the hardware.

The timer uses the value of **CPU clock** to calculate the time for each interrupt. For example, a model with a sine wave generator block running at 1 kHz uses timer interrupts to generate sine wave samples at the proper rate. For example, using 100 MHz, the timer calculates the sine generator interrupt period as follows:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

100,000,000/1000 = 1 Sine block interrupt per 100,000 clock ticks

Board Support

Select the following parameters and edit their values in the text box on the right:

- `Source files` — Enter the full paths to source code files.
- `Include paths` — Add paths to include files.
- `Libraries` — Identify specific libraries for the processor. Required libraries appear on the list by default. To add more libraries, entering the full path to the library with the library file in the text area.
- `Initialize functions` — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.

- `Terminate functions` — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

Note Invalid or incorrect entries in these fields can cause errors during code generation. When you enter a file path, library, or function, the block does not verify that the path or function exists or is valid.

When entering a path to a file, library, or other custom code, use the following string in the path to refer to the IDE installation folder.

```
$(Install_dir)
```

Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code. **Support** options do not support functions that use return arguments or values. These parameters accept only functions of type `void fname void` as valid as entries.

You can also set up environment variables to use as folder path tokens. For example, if you set up an environment called `USER_VAR`, you can use it as a token when you define a path in your Target Preferences block. For example: `$(USER_VAR)\myinstal\foo.c`.

Operating System

Select an operating system or RTOS for your target. If your target platform supports an operating system, the software enables the **Operating system** parameter. Otherwise, the software disables this option.

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 ensure that the Target Preferences block, the IDE, and the hardware board all refer to the same processor. Otherwise, during code generation, the software generates a warning similar to the following message:

```
Target Preferences block specifies that the board named '<boar
will be used to run generated code.
However, since only board named '<boardname2>' is found in you
that board will be used.
```

Board Name

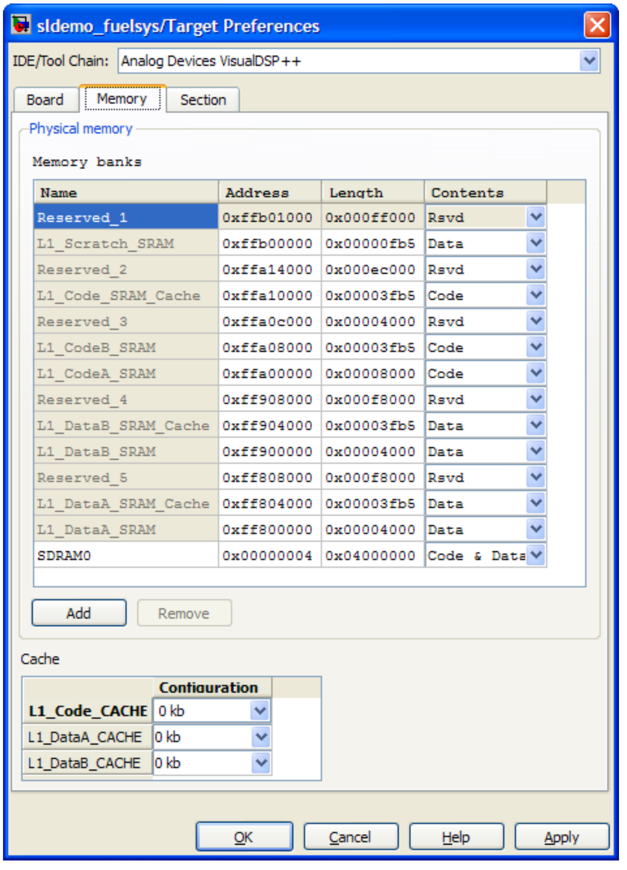
Board Name appears after you click **Get from IDE**. Select the board you are using. Match **Board Name** with the **Board** option near the top of the **Board** pane.

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**.

Memory Pane



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. For supported boards, the board-specific Target Preferences blocks set the default memory map.

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 Preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured Target Preferences blocks shows the memory segments available on the board, but external to the processor. Target Preferences blocks 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)

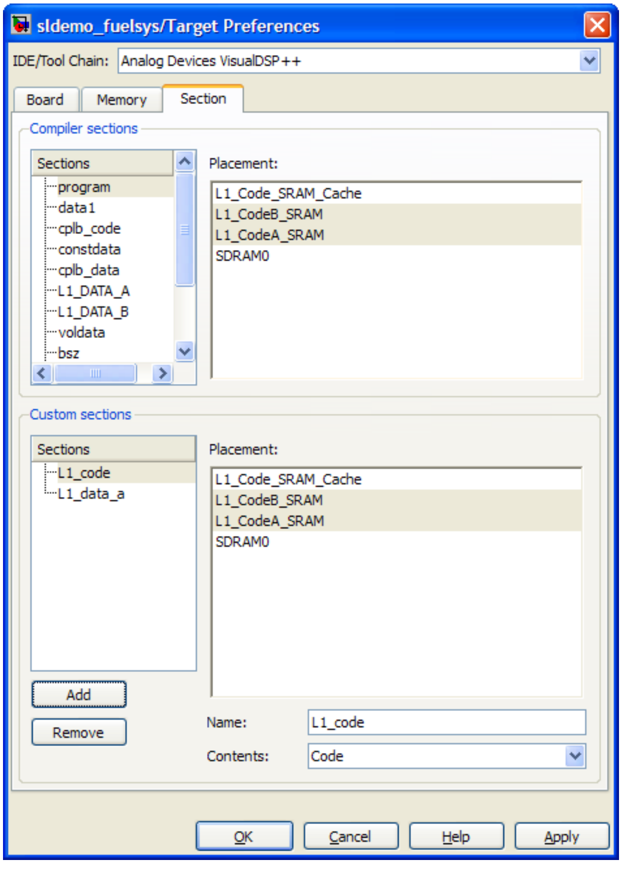
When the **Processor** on the Board pane supports a cache memory structure, the dialog box displays a table of **Cache** parameters. You can use this table to configure the cache as SRAM and partial cache. Both the data memory and the program share this second-level memory.

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`.

Section Pane



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
`.bss`	Compiler	Static and global C variables in the code
`.cinit`	Compiler	Tables for initializing global and static variables and constants
`.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
`.sysmem`	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 any 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`.

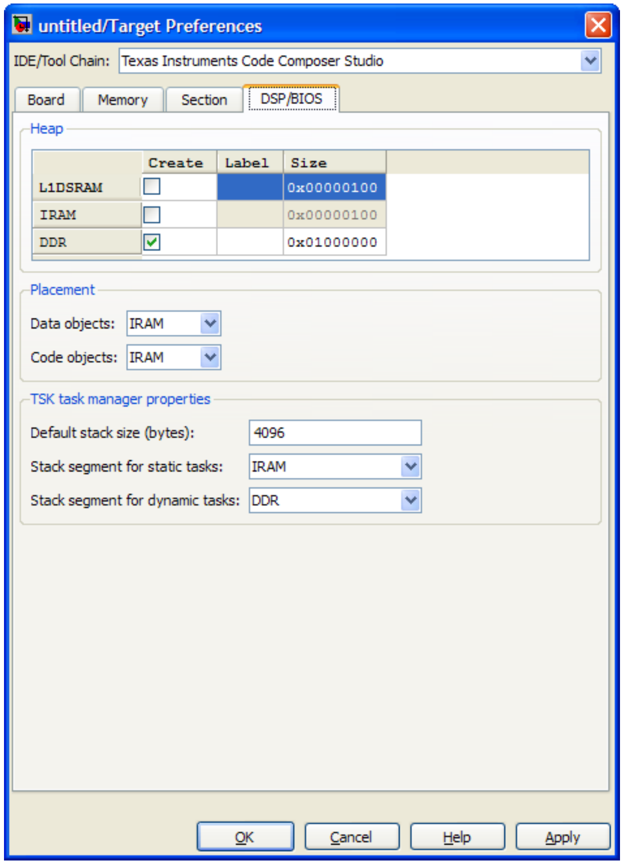
Contents
Identify whether the contents of the new section are `Code`, `Data`, or `Any`.

Remove
To remove a section from the **Custom sections** list, select the section and click **Remove**.

DSP/BIOS Pane

The DSP/BIOS pane is available if the two following conditions are true:

- You are using Texas Instruments CCS IDE.
- You set the Target Preferences block **Processor** option to a C6000 processors that support 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. You can set any size up to the limits for 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.

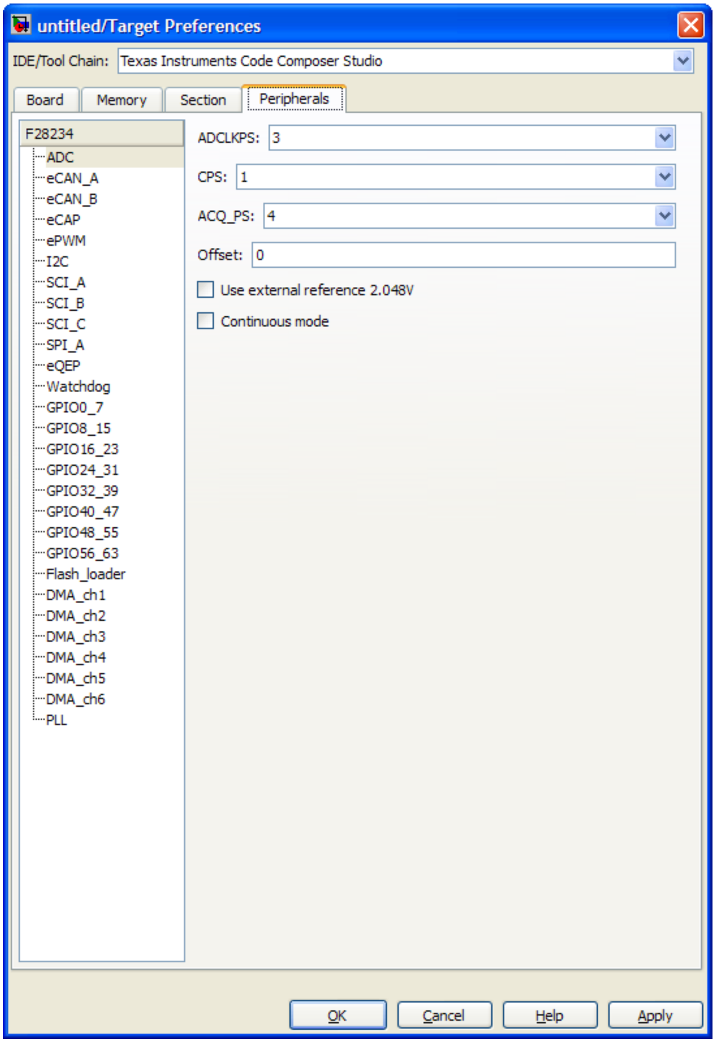
Stack segment for static tasks
Use this option to specify where to allocate the stack for static tasks. Tasks that your program uses often are good candidates for static tasks. Infrequently used tasks usually work best as dynamic tasks.

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 "Memory Pane" on page 5-868.

Peripherals Pane



The Peripherals pane is only visible in Target Preference blocks configured for C2000 processors. This tabbed pane appears to configure peripheral settings and pin assignments.

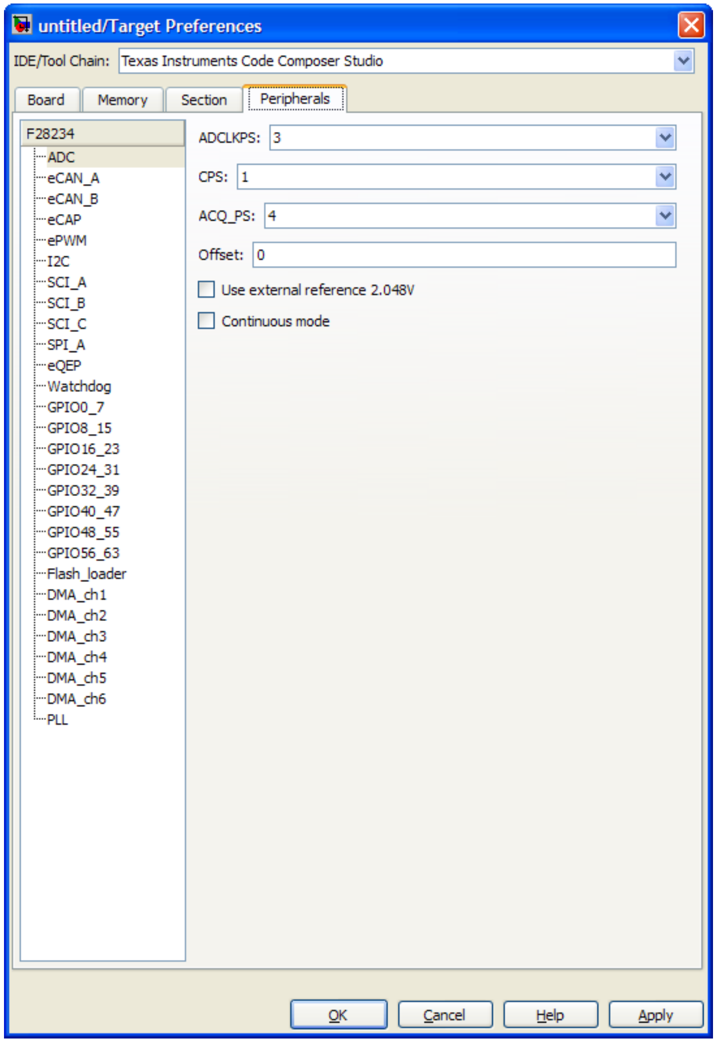
To set the attributes for a peripheral, select the peripheral from the **Peripherals** list and then set the attribute options on the right side.

The following table describes all the peripherals provided on the **Peripherals** list. Some peripherals are not available on some C2000 processors.

Peripheral Name	Description
"ADC" on page 5-883	Analog-to-Digital Converter (ADC) parameters
"eCAN_A, eCAN_B" on page 5-886	Enhanced Controller Area Network (eCAN) parameters for modules A or B
"eCAP" on page 5-889	Enhanced Capture (eCAP) parameters for pin mapping to GPIO
"ePWM" on page 5-891	Enhanced Pulse Width Modulation (ePWM) parameters for pin mapping to GPIO
"I2C" on page 5-893	Inter-Integrated Circuit (I2C) parameters for communications
"SCI_A, SCI_B, SCI_C" on page 5-900	Serial Communications Interface (SCI) parameters for communications with modules A, B, or C
"SPI_A, SPI_B, SPI_C, SPI_D" on page 5-904	Serial Peripheral Interface (SPI) parameters for communications with module A, B, C, or D

Peripheral Name	Description
"eQEP" on page 5-907	Enhanced Quadrature Encoder Pulse (eQEP) parameters for pin mapping to GPIO
"Watchdog" on page 5-909	Watchdog enable/disable and timing
"GPIO" on page 5-911	General Purpose Input Output (GPIO) parameters for input qualification types
"Flash_loader" on page 5-915	Flash memory loader/programmer
"DMA_ch[#]" on page 5-917	Direct Memory Access (DMA) parameters for channels 1 to N
"PLL" on page 5-932	Phase Loop Lock (PLL) parameters to adjust clock settings and match custom oscillator frequencies
"LIN" on page 5-934	Local Interconnect Network (LIN) parameters for communications

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 scalers, refer to "Configuring ADC Parameters for Acquisition Window Width".

You can set the following parameters for the ADC clock prescaler:

ACQ_PS
: This value does not actually have a direct effect on 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.

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.

CPS
: After dividing the HSPCLK speed by the **ADCLKPS** value, setting the **CPS** parameter to 1, the default value, divides the result by 2.

Use external reference 2.048V
: By default, an internally generated band gap voltage reference supplies the ADC logic. However, depending on application requirements, you can enable **External reference** so the ADC logic uses an external voltage reference instead. Select the checkbox to use a 2.048V external voltage reference.

Offset
: 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
: (For Piccolo processors) When you disable the **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

(For Piccolo processors) 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

(For Piccolo processors) 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.

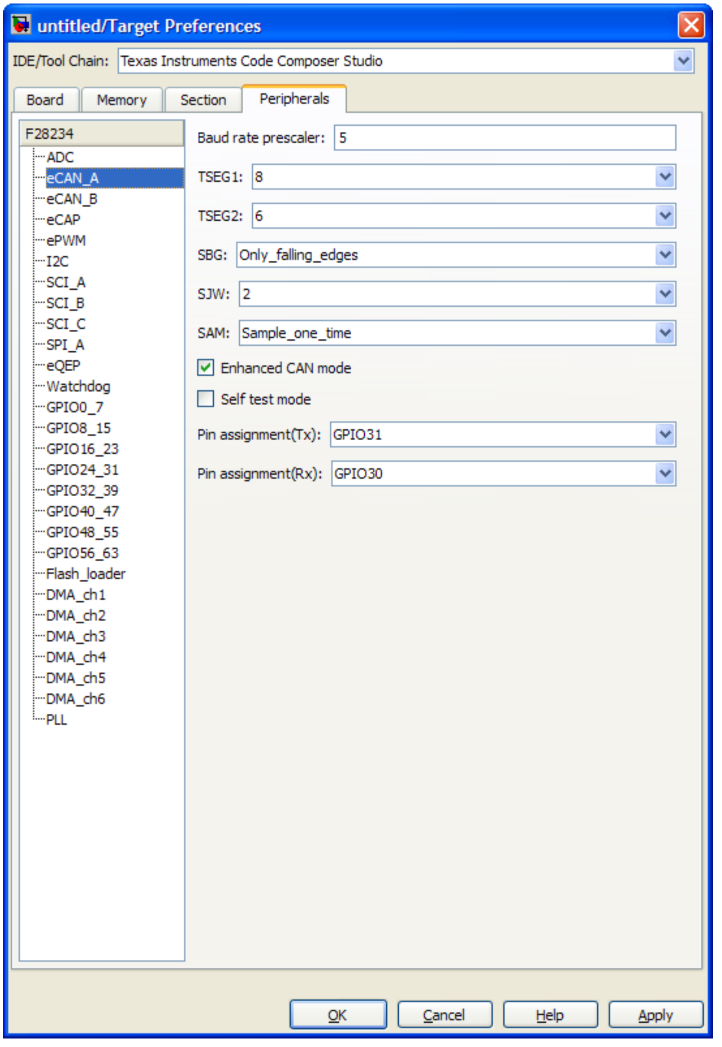
Choose one of the `high priority` selections to assign high priority to one or more of the SOCs. In this mode, the ADC operates in round robin mode until it receives a high priority SOC interrupt. The ADC finishes servicing the current SOC, services the high priority SOCs, and then returns to the next 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

(For Piccolo processors) Select the pin to which the ADC sends the XINT2SOC pulse.

eCAN_A, eCAN_B


untitled/Target Preferences
IDE/Tool Chain: Texas Instruments Code Composer Studio
Board
Memory
Section
Peripherals
F28234
ADC
eCAN_A
eCAN_B
eCAP
ePWM
I2C
SCI_A
SCI_B
SCI_C
SPI_A
eQEP
Watchdog
GPIO0_7
GPIO8_15
GPIO16_23
GPIO24_31
GPIO32_39
GPIO40_47
GPIO48_55
GPIO56_63
Flash_loader
DMA_ch1
DMA_ch2
DMA_ch3
DMA_ch4
DMA_ch5
DMA_ch6
PLL
Baud rate prescaler: 5
TSEG1: 8
TSEG2: 6
SBG: Only_falling_edges
SJW: 2
SAM: Sample_one_time
Enhanced CAN mode
Self test mode
Pin assignment(Tx): GPIO31
Pin assignment(Rx): GPIO30
OK
Cancel
Help
Apply

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:

Baud rate prescaler
: Value by which to scale the bit rate. Valid values are from `1` to `256`.

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".

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.

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.

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`.

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`.

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.

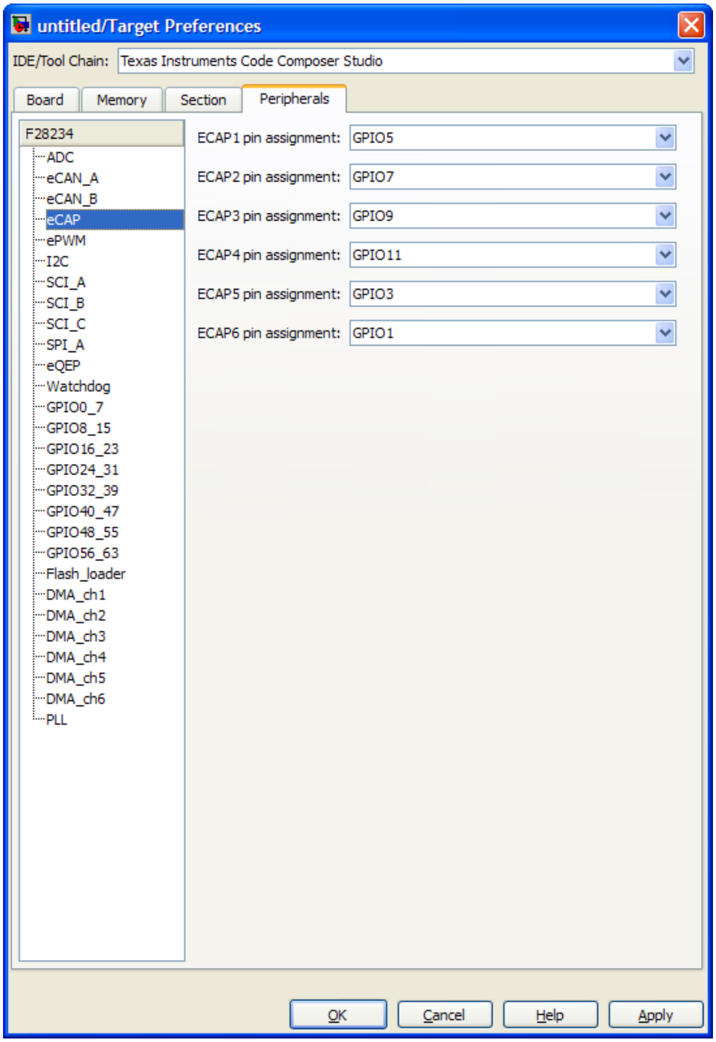
Pin assignment (Rx)

Assigns the CAN receive pin to use with the eCAN_B module. Possible values are GPIO10, GPIO13, GPIO17, and GPIO21.

Pin assignment (Tx)

Assigns the CAN transmit pin to use with the eCAN_B module. Possible values are GPIO8, GPIO12, GPIO16, and GPIO20.

eCAP



Assigns eCAP pins to GPIO pins if necessary.

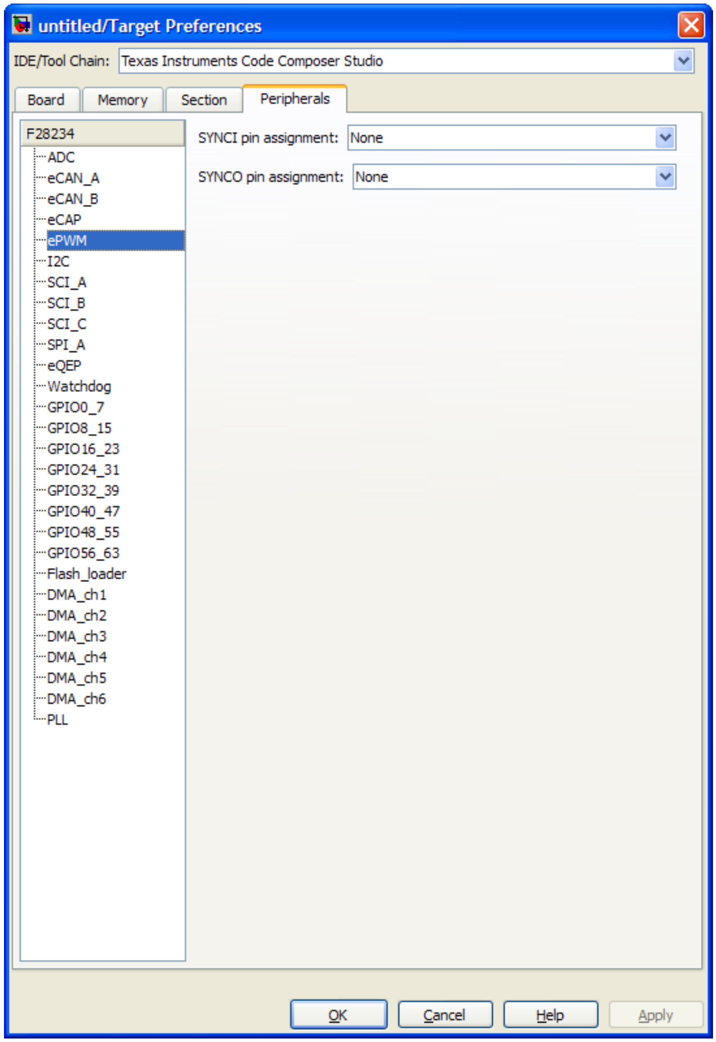
ECAP1 pin assignment
Select an option from the list—`None`, `GPIO5`, or `GPIO24`.

ECAP2 pin assignment
Select an option from the list—`None`, `GPIO7`, or `GPIO25`.

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`.

ePWM



Assigns ePWM signals to GPIO pins, if necessary.

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
Assigns the ePWM external sync pulse output (SYNCO) to a GPIO pin. Choices are `None` (the default), `GPIO6`, and `GPIO33`.

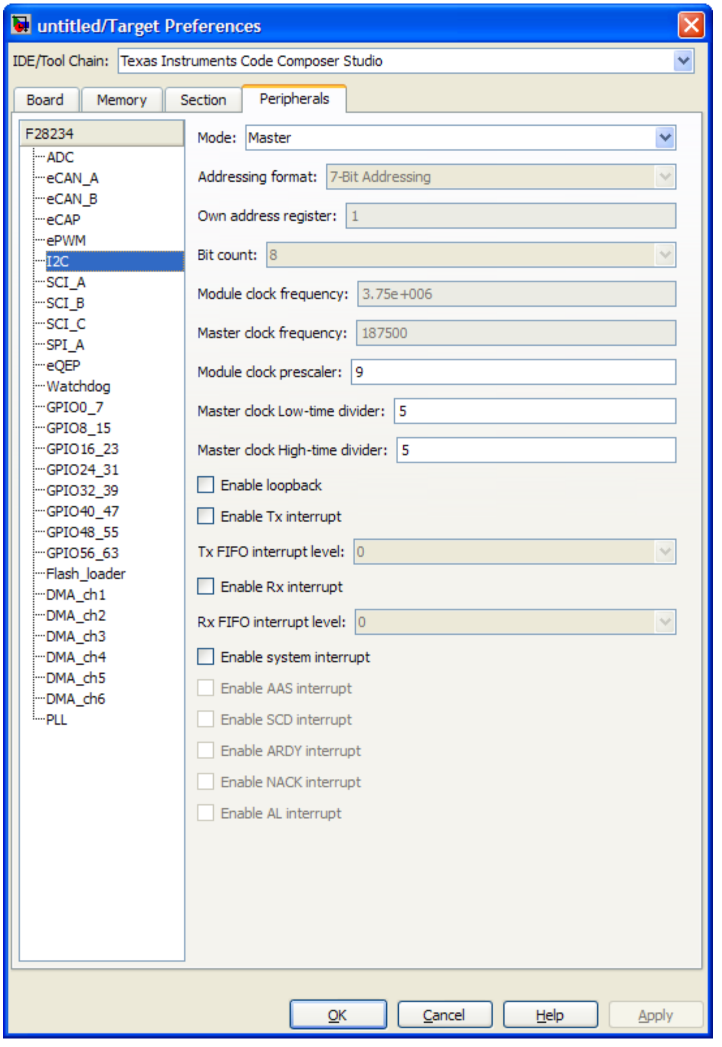
TZ2 pin assignment
Assigns the trip-zone input 2 (TZ2) to a GPIO pin. Choices are `None` (the default), `GPIO16`, and `GPIO28`.

TZ3 pin assignment
Assigns the trip-zone input 3 (TZ3) to a GPIO pin. Choices are `None` (the default), `GPIO17`, and `GPIO29`.

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`.

I2C



Report or set Inter-Integrated Circuit parameters. For more information, consult the *TMS320x280x Inter-Integrated Circuit Module*

Reference Guide, Literature Number: SPRU721A, 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 (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 frequency

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, available on the Texas Instruments Web site.

Master clock frequency

This field displays the master clock frequency. For more information about this value, consult the "Clock Generation" section in the *TMS320x280x Inter-Integrated Circuit Module Reference Guide*, Literature Number: SPRU721, available on the Texas Instruments Web site.

Module clock prescaler

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** corresponds to bits 7–0 (IPSC) of the I2C Prescaler Register (I2CPSC).

Master clock Low-time divider

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 = Tmod x (ICCL + d).

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: SPRU721A, 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

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 = Tmod x (ICCL + d).

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: SPRU721A, 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 (I2CMDR).

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).

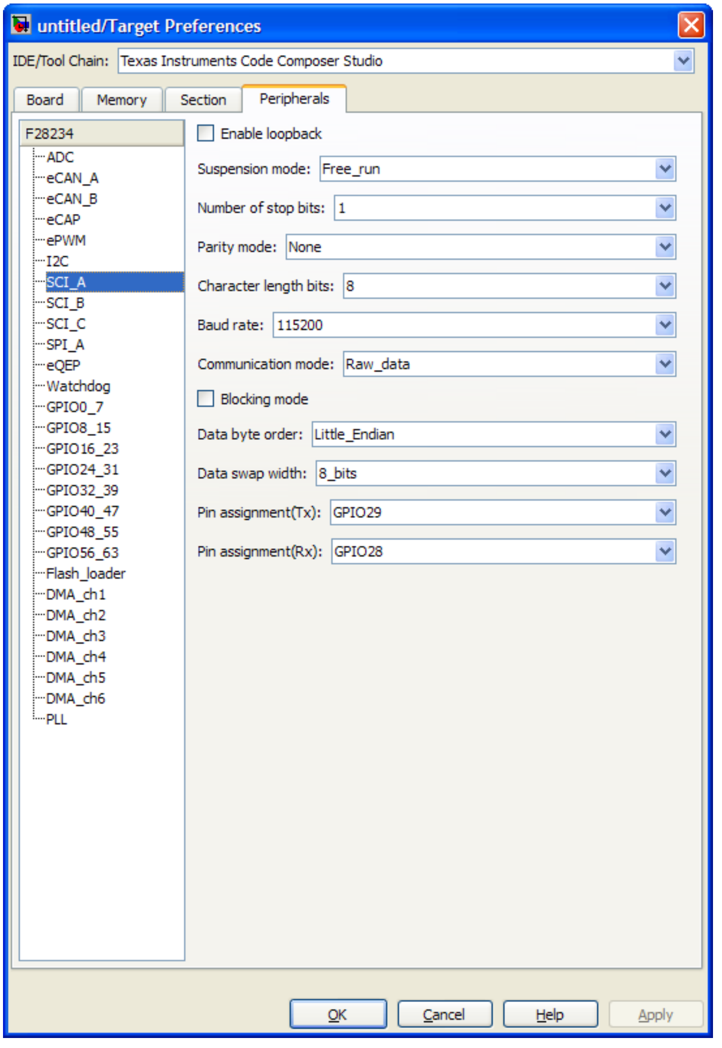
Enable AL interrupt

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 looses 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:

Baud rate

Baud rate for transmitting and receiving data. Select from 115200 (the default), 57600, 38400, 19200, 9600, 4800, 2400, 1200, 300, and 110.

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) to see if there is any data to read. If data is present, it reads and outputs the contents. If no data is present, it outputs the last value and continues.

Character length bits

Length in bits of each transmitted or received character, set to 8 bits.

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. No deadlock condition can occur because there is no wait state. 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 any 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
- Ensures that data is received correctly (checksum)
- Ensures that data is received by processor

- Ensures 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.

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`.

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.

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.

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.

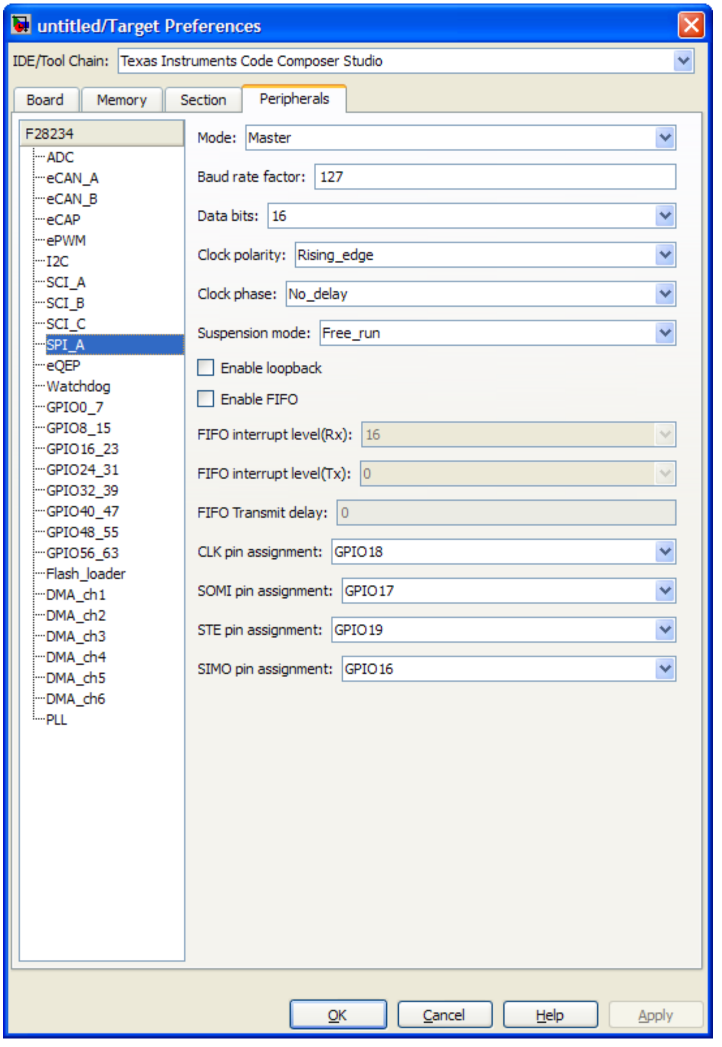
Pin assignment (Rx)

Assigns the SCI receive pin to use with the SCI module.

Pin assignment (Tx)

Assigns the SCI transmit 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:

Baud rate factor
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.

Clock phase
Select `No_delay` or `Delay_half_cycle`.

Clock polarity
Select `Rising_edge` or `Falling_edge`.

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.

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.

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.

Mode
Set to `Master` or `Slave`.

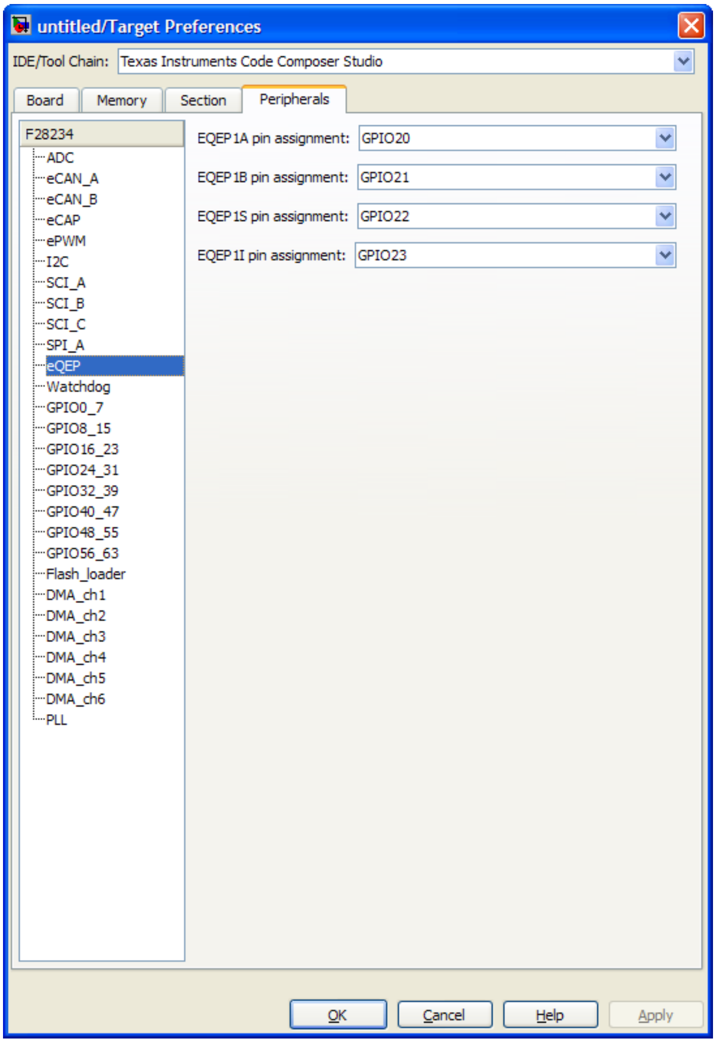
CLK pin assignment
Assigns the SPI something (CLK) to a GPIO pin. Choices are `None` (default), `GPIO14`, or `GPIO26`.

SOMI pin assignment
Assigns the SPI something (SOMI) to a GPIO pin. Choices are `None` (default), `GPIO13`, or `GPIO25`.

STE pin assignment
Assigns the SPI something (STE) to a GPIO pin. Choices are `None` (default), `GPIO15`, or `GPIO27`.

SIMO pin assignment
Assigns the SPI something (SIMO) to a GPIO pin. Choices are `None` (default), `GPIO12`, or `GPIO24`.

eQEP



Assigns eQEP pins to GPIO pins.

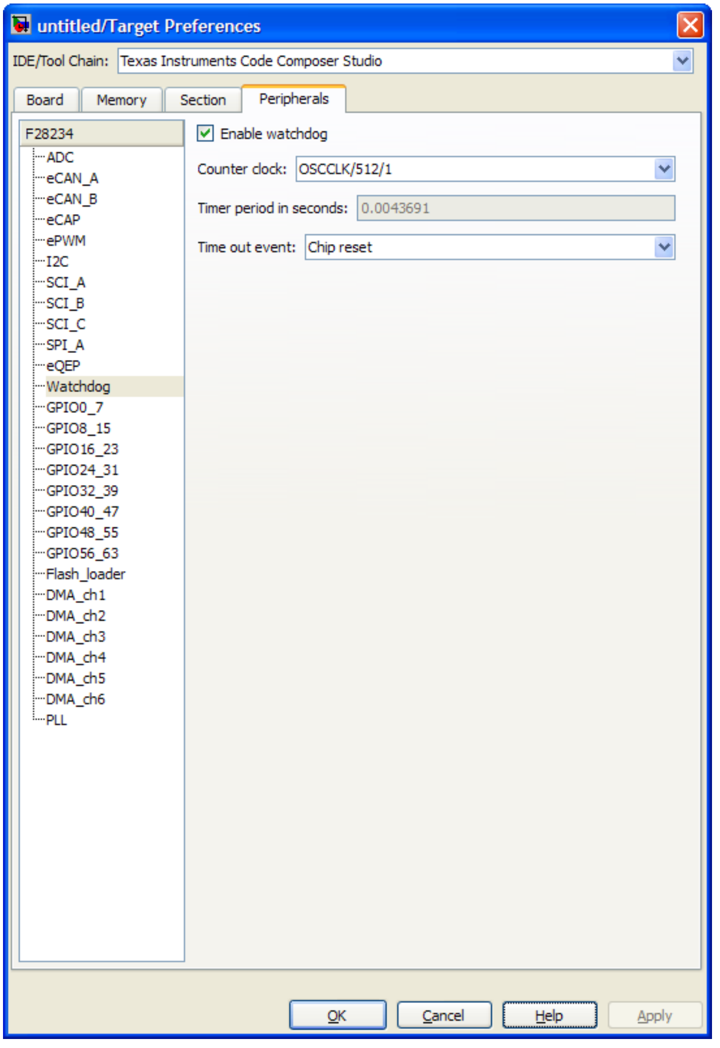
EQEP1A pin assignment
Select an option from the list—`GPIO20` or `GPIO50`.

EQEP1B pin assignment
Select an option from the list—`GPIO21` or `GPIO51`.

EQEP1S pin assignment
Select an option from the list—`GPIO22` or `GPIO52`.

EQEP1I pin assignment
Select an option from the list—`GPIO23` or `GPIO53`.

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).

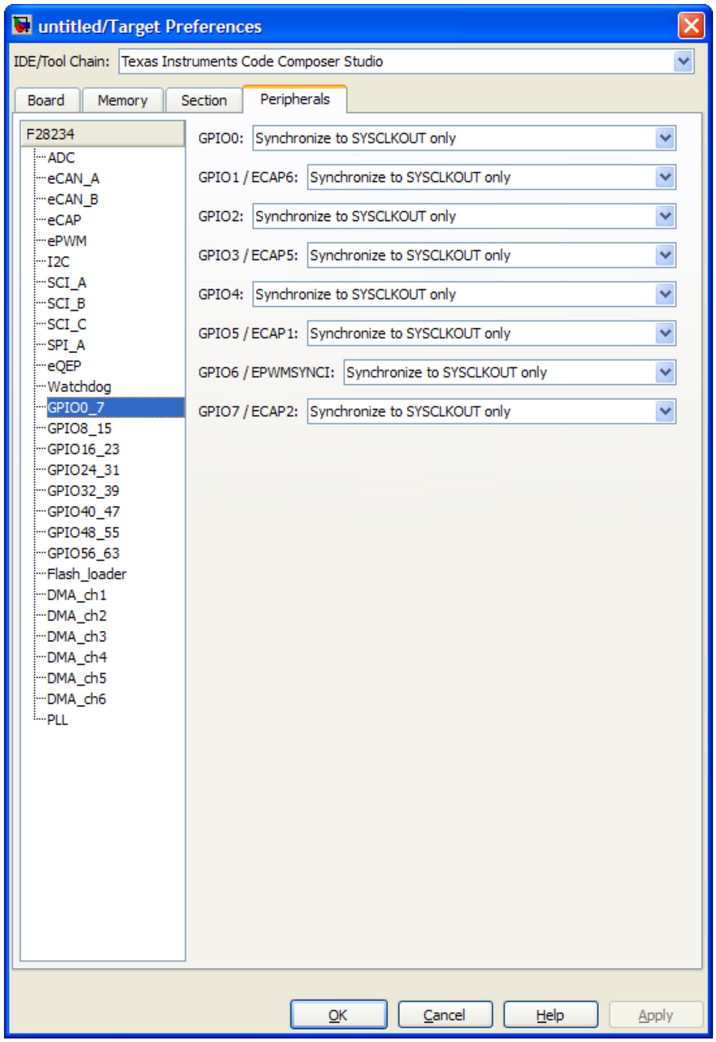
Timer period in seconds
This field displays the timer period in seconds. This value automatically updates when you change the **Counter clock** parameter.

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.

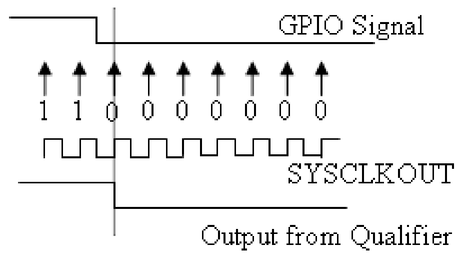
This parameter corresponds to bit 1 (WDENINT) of the System Control and Status Register (SCSR).

GPIO

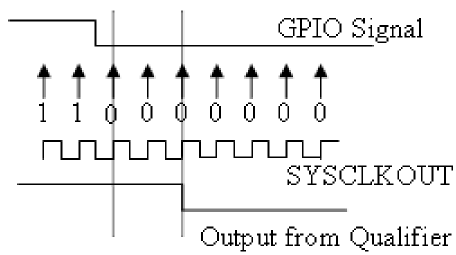


Each pin selected for input offers three signal qualification types:

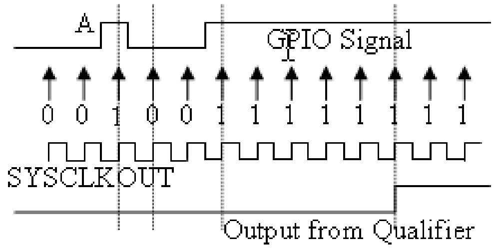
- Sync to SYSCLKOUT — This setting is the default for all pins at reset. Using this qualification type, the input signal is synchronized to the system clock SYSCLKOUT. The following figure shows the input signal measured on each tick of the system clock, and the resulting output from the qualifier.



- 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.



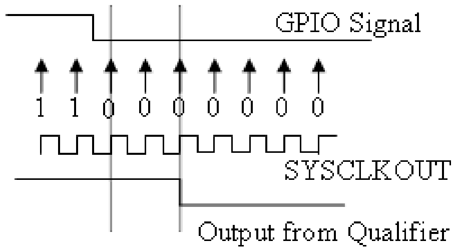
- Qualification using 6 samples — This setting requires six consecutive cycles of the same GPIO input value for the output from the qualifier to change. In the following figure, the glitch **A** has no effect on the output signal. When the glitch occurs, the counting begins, but the next measurement is low again, so the count is ignored. The output signal does not change until six consecutive samples of the high signal are measured.



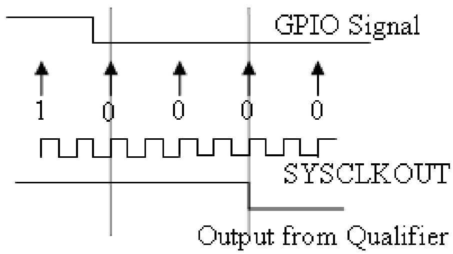
Qualification sampling period prescaler

Visible only when an appropriate 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 SYSCLKOUT/(2 * 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.

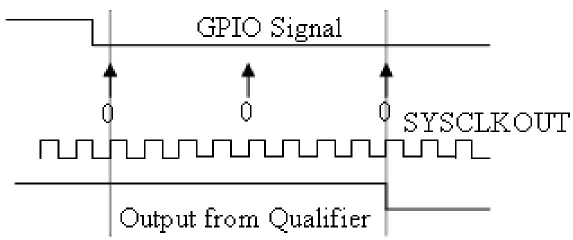
The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to `Qualification using 3 samples`. In this case, the **Qualification sampling period prescaler**=`0`:



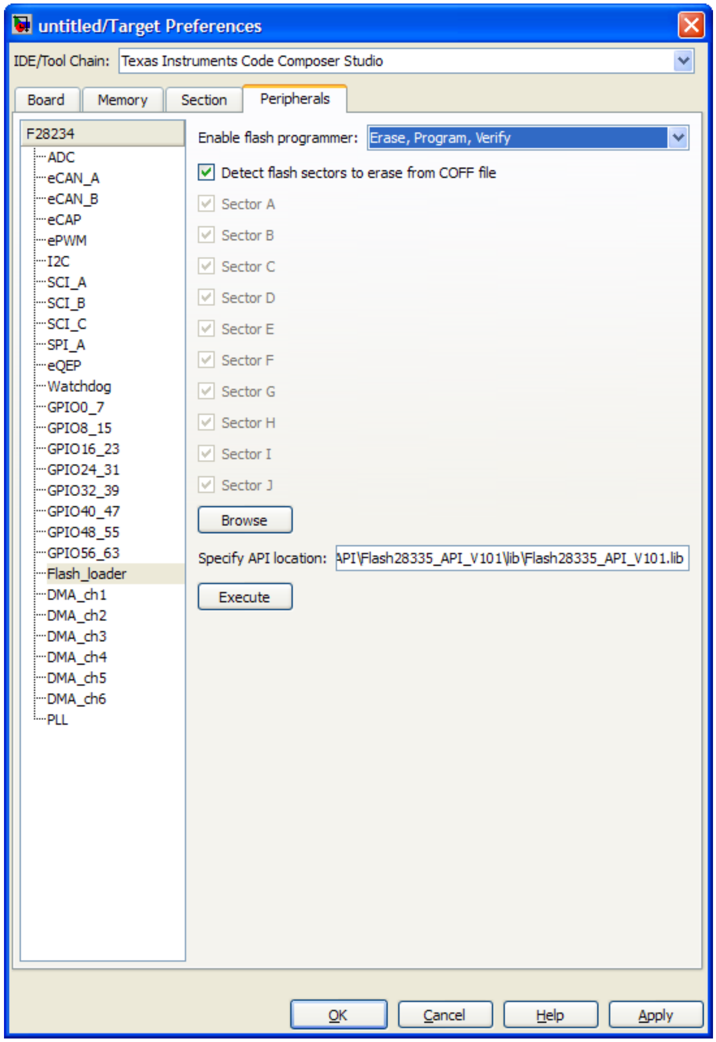
In the next figure **Qualification sampling period prescaler**=`1`. A sample is taken every two clock ticks, and the **Qualification type** is set to `Qualification using 3 samples`. The output signal changes much later than if **Qualification sampling period prescaler**=`0`.



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`.



Flash_loader



You can use Flash_loader to:

- Automatically 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 appropriate 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.

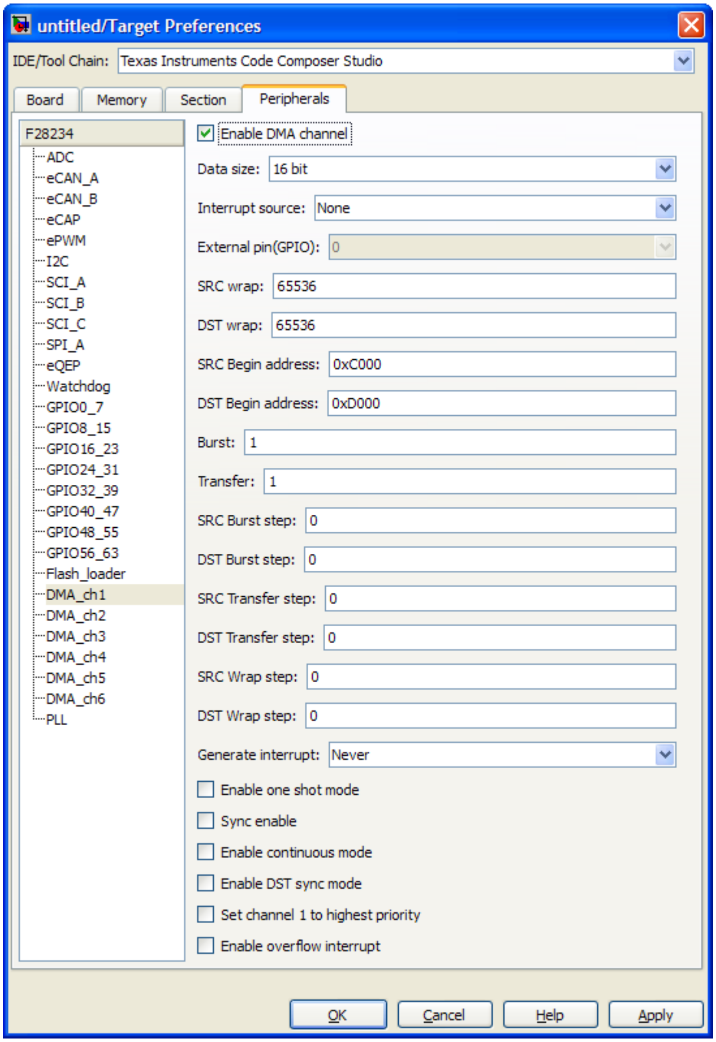
Enable Flash Programmer
: Enable the flash programmer by selecting a task for it to perform when you click **Execute** or build the software. To program the flash memory when you build the software, select `Erase, Program, Verify`.

Detect Flash sectors to erase from COFF file
: When enabled, the flash programmer erases all of the flash sectors defined by the COFF file.

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.

Execute
: Click this button to initiate the task selected in **Enable Flash Programmer**.

DMA_ch[#]


untitled/Target Preferences
IDE/Tool Chain: Texas Instruments Code Composer Studio
Board
Memory
Section
Peripherals
F28234
ADC
eCAN_A
eCAN_B
eCAP
ePWM
I2C
SCI_A
SCI_B
SCI_C
SPI_A
eQEP
Watchdog
GPIO0_7
GPIO8_15
GPIO16_23
GPIO24_31
GPIO32_39
GPIO40_47
GPIO48_55
GPIO56_63
Flash_loader
DMA_ch1
DMA_ch2
DMA_ch3
DMA_ch4
DMA_ch5
DMA_ch6
PLL
Enable DMA channel
Data size: 16 bit
Interrupt source: None
External pin(GPIO): 0
SRC wrap: 65536
DST wrap: 65536
SRC Begin address: 0xC000
DST Begin address: 0xD000
Burst: 1
Transfer: 1
SRC Burst step: 0
DST Burst step: 0
SRC Transfer step: 0
DST Transfer step: 0
SRC Wrap step: 0
DST Wrap step: 0
Generate interrupt: Never
Enable one shot mode
Sync enable
Enable continuous mode
Enable DST sync mode
Set channel 1 to highest priority
Enable overflow interrupt
OK
Cancel
Help
Apply

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 use DMA with the C280x/C28x3x ADC block, open the ADC block, enable **Use DMA (with C28x3x)**, and select a DMA channel number. To avoid error messages, open the **Target Preferences block > Peripherals** and *disable* the same DMA channel number.

For more information, consult the *TMS320x2833x, 2823x Direct Memory Access (DMA) Module Reference Guide*, Literature Number: SPRUFB8A, and 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.

If your model includes an ADC block with the **Use DMA (with C28x3x)** parameter enabled, disable the same DMA channel here in the Target Preferences block.

This parameter has no 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).

Note When you select **Use DMA (with C28x3x)** in the ADC block, this parameter is `16 bit`.

Interrupt source

Select the peripheral interrupt that triggers a DMA burst for the specified channel.

Selecting SEQ1INT or SEQ2INT generates a message: "Use ADC block to implement the DMA function." To do so, open the ADC block, select the **Use DMA (with C28x3x)** parameter, select a DMA channel, and disable the same DMA channel in the Target Preferences block. Currently, when you use the ADC block to implement DMA, the corresponding DMA channel settings are not configurable in the Target Preferences block.

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.
- The C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input and C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output block reference sections.

Currently, **Interrupt source** does not support items TINT0 through MREVTB in the drop-down menu.

The **Interrupt source** parameter corresponds to bit 4-0 (PERINTSEL) in the Mode Register (MODE).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block:

- If the ADC block **Module** is `A` or `A and B`, **Interrupt source** is `SEQ1INT`.
- If the ADC block **Module** is `B`, **Interrupt source** is `SEQ2INT`.

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 System Control and*

Interrupts Reference Guide, Literature Number SPRUFB0, 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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this parameter is 65536.

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this parameter is 65536.

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of the source **Begin address** is:

- 0xB00 if the ADC block **Module** is `A` or `A and B` (**Interrupt source** is `SEQ1INT`).
- 0xB08 If the ADC block **Module** is `B` (**Interrupt source** is `SEQ2INT`).

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of the destination **Begin address** (dstAdd) is the ADC buffer address (ADCbufadr) minus the **Number of conversions** (NoC) in the ADC block. In other words, dstAdd = ADCbufadr – NoC.

- If the target is F28232 or F28332, ADCbufadr = 57340 (0xDFFC)
- Otherwise, ADCbufadr = 65532 (0xFFFC)

For example, when you enable **Use DMA (with C28x3x)** for a F28232 target, the DMA module sets the destination **Begin address** to 0xDFF9 (57337) because the ADCbufadr 57340 (0xDFFC) minus 3 conversions equals 57337 (0xDFF9).

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 appropriately 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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value assigned to **Burst** equals the ADC block **Number of conversions** (NOC) multiplied by a value for the ADC block **Conversion mode** (CVM). Burst = NOC * CVM

If **Conversion mode** is `Sequential`, CVM = 1. If **Conversion mode** is `Simultaneous`, CVM = 2.

For example, Burst = 6 if NOC = 3 and CVM = 2 (6 = 3 * 2).

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this parameter is 1.

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 correct 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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is 1.

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 correct 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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is 1.

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** has no effect.

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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this parameter is 0.

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** has no effect.

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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this destination parameter is 1.

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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this parameter is 0.

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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the value of this parameter is 0.

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, the DMA channel generates an interrupt at the end of the data transfer.

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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is disabled.

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 ensures that the wrap counter and the ADC channels remain synchronized with each other.

If **Interrupt source** is not set to SEQ1INT, **Sync enable** has no effect.

This parameter corresponds to bit 12 (SYNCE) of the Mode Register (MODE).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is disabled.

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.

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is enabled.

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is disabled.

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is disabled.

Enable overflow interrupt

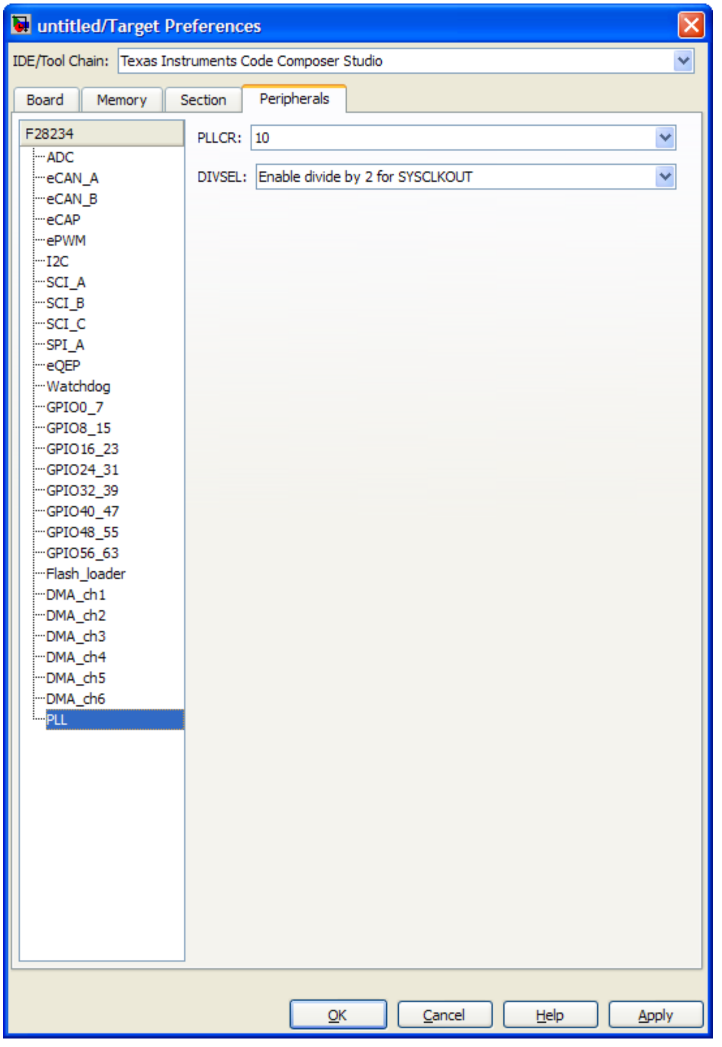
Enable this parameter to have the DMA channel send an interrupt to the CPU via PIE if the DMA module receives a peripheral interrupt while a previous interrupt from the same peripheral is waiting to be serviced.

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).

Note When you select **Use DMA (with C28x3x)** in the C280x/C28x3x ADC block, this parameter is disabled.

PLL



The default PLL register values run the CPU clock (CLKIN) at its maximum frequency. The parameters assume that the external oscillator frequency on the board (OSCCLK) is the one recommended by the processor vendor.

Change the PLL settings if:

- You want to change the CPU frequency.
- The external oscillator frequency differs from the value recommended by the manufacturer.

Use the following equation to determine the CPU frequency (CLKIN):

CLKIN = (OSCCLK * PLLCR) / (DIVSEL or CLKINDIV)

Where:

- 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”.

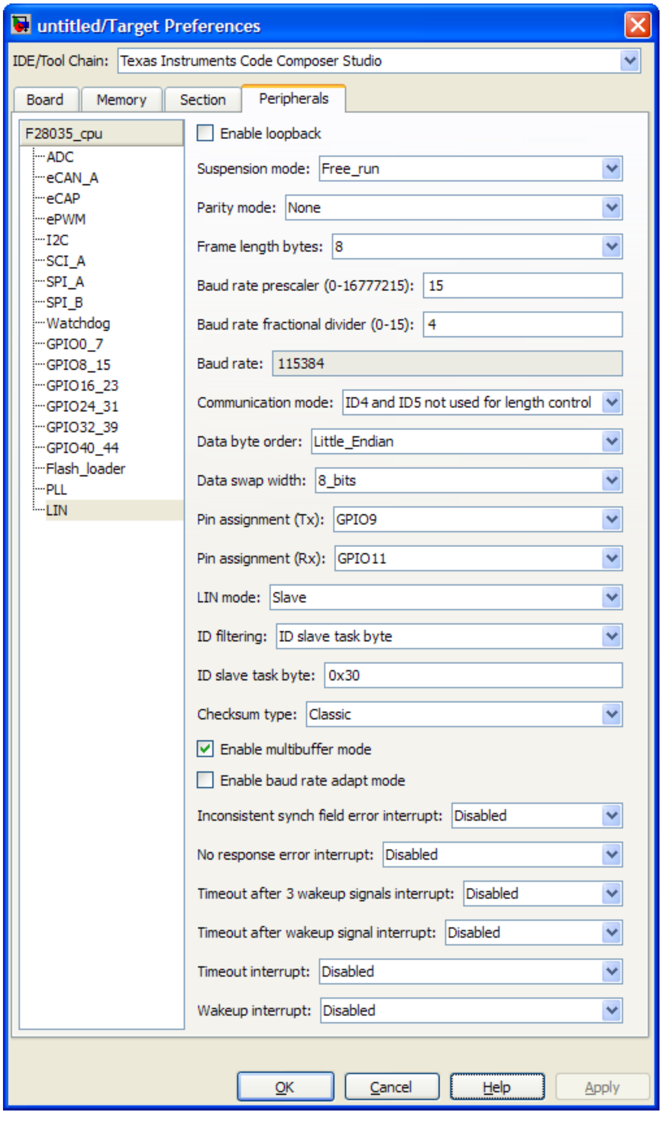
The availability of the DIVSEL or CLKINDIV parameters change depending on the selected processor. If neither parameter is available, use the following equation instead:

CLKIN = (OSCCLK * PLLCR) / 1

Enter the resulting CPU clock frequency (CLKIN) in the **CPU clock** parameter of the Target Preferences block.

For more information, consult the “PLL-Based Clock Module” section in the Texas Instruments *Reference Guide* for your processor.

LIN


untitled/Target Preferences
IDE/Tool Chain: Texas Instruments Code Composer Studio
Board
Memory
Section
Peripherals
F28035_cpu
ADC
eCAN_A
eCAP
ePWM
I2C
SCI_A
SPI_A
SPI_B
Watchdog
GPIO0_7
GPIO8_15
GPIO16_23
GPIO24_31
GPIO32_39
GPIO40_44
Flash_loader
PLL
LIN
Enable loopback
Suspension mode: Free_run
Parity mode: None
Frame length bytes: 8
Baud rate prescaler (0-16777215): 15
Baud rate fractional divider (0-15): 4
Baud rate: 115384
Communication mode: ID4 and ID5 not used for length control
Data byte order: Little_Endian
Data swap width: 8_bits
Pin assignment (Tx): GPIO9
Pin assignment (Rx): GPIO11
LIN mode: Slave
ID filtering: ID slave task byte
ID slave task byte: 0x30
Checksum type: Classic
Enable multibuffer mode
Enable baud rate adapt mode
Inconsistent synch field error interrupt: Disabled
No response error interrupt: Disabled
Timeout after 3 wakeup signals interrupt: Disabled
Timeout after wakeup signal interrupt: Disabled
Timeout interrupt: Disabled
Wakeup interrupt: Disabled
OK
Cancel
Help
Apply

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.

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
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
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
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 never reports matches.

I you select `ID slave task`. the module uses the RECID an ID-SlaveTask byte to detect a match. If you select this option and enter 0xFF for LINMASK, the LIN module always 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 appropriate 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 correctly, 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 an appropriate timeframe.

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

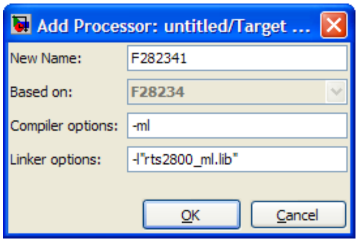
The dialog box displays this option when you set **LIN mode** to `Slave`.

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 any valid C string. The name you enter in this field appears on the list of processors after you add the new processor.

If you do not provide an entry for each parameter, the coder product returns an error message without creating a processor entry.

Based On

When you add a processor, the dialog box uses the settings from the currently selected processor as the basis for the new one. This parameter displays the currently selected processor.

Compiler options

Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class.

For example, to set the compiler switch for a new C5509 processor, enter `-ml`. The following table shows the compiler switch string for supported processor families.

Processor Family	Compiler Switch String
C62xx	None
C64xx	None
C67xx	None
DM64x and DM64xx	None

Processor Family	Compiler Switch String
C55xx	`-ml`
C28xx, F28xx, R28xx, F28xxx	`-ml`

Linker options

You can use this parameter to specify linker command options. The IDE uses these options to modify how it links project files when you build a project. To get information about specific linker options you can enter here, consult the documentation for your IDE.

Linux Pane

The Linux tab appears when you set **IDE/Tool Chain** to `Eclipse` and set **Operating System** on the Board tab to `Linux`.

The Linux tab displays two options:

Scheduling Mode

When you select `free-running`, the model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.

When you select `real-time`, the model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a Linux real-time clock.

Base rate task priority

The base rate in the model maps to a thread and runs as fast as possible. You can use the value of the base rate priority to set a static priority for the base rate task. By default, this rate is 40.

Allow tasks to execute concurrently

> **Note** This parameter will be removed in a future release.

Enable multicore deployment. Selecting this option enables generated multi-threading code to run concurrently on multicore processors. By default, this option is disabled.

This parameter has been superseded. Configuring the model as described in the following procedures hides the **Allow tasks to execute concurrently** parameter from view.

To run target applications on multicore processors, follow the procedures in "Running Target Applications on Multicore Processors", and "Configuring Models for Targets with Multicore Processors".

VxWorks Pane

The VxWorks tab appears when you set **IDE/Tool Chain** to `Wind River Diab/GCC (makefile generation only)` and set **Operating System** on the Board tab to `VxWorks`.

The Linux tab displays two options:

Scheduling Mode

When you select `free-running`, the model generates multi-threaded free-running code. Each rate in the model maps to a separate thread in the generated code. Multi-threaded code can potentially run faster than single threaded code.

When you select `real-time`, the model generates multi-threaded real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-second rate runs at exactly 1-second intervals. The timing is provided by using a Linux real-time clock.

Base rate task priority

The base rate in the model maps to a thread and runs as fast as possible. You can use the value of the base rate priority to set a static priority for the base rate task. By default, this rate is 40.

Allow tasks to execute concurrently

Note This parameter will be removed in a future release.

Enable multicore deployment. Selecting this option enables generated multi-threading code to run concurrently on multicore processors. By default, this option is disabled.

This parameter has been superseded. Configuring the model as described in the following procedures hides the **Allow tasks to execute concurrently** parameter from view.

To run target applications on multicore processors, follow the procedures in “Running Target Applications on Multicore Processors”, and “Configuring Models for Targets with Multicore Processors”.

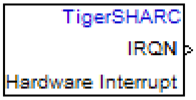
TigerSHARC Hardware Interrupt

Purpose

Generate Interrupt Service Routine

Library

Embedded Coder/ Embedded Targets/ Processors/ Analog Devices TigerSHARC/ Scheduling

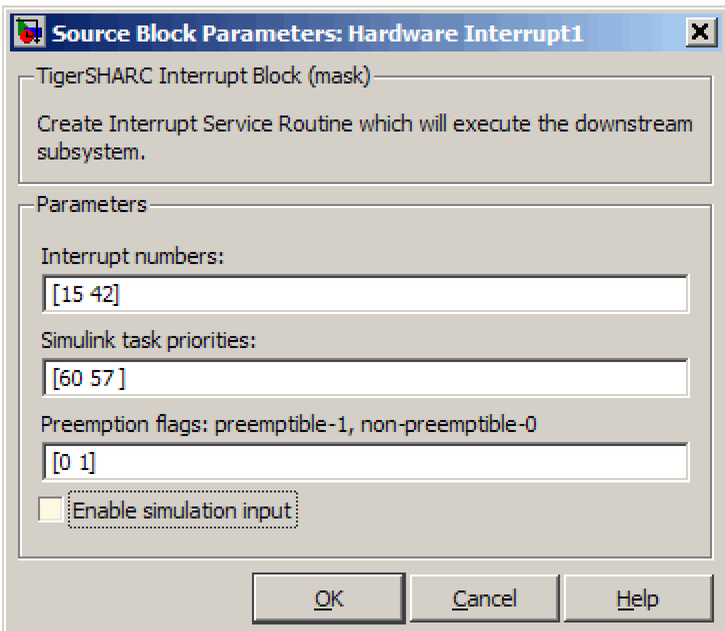


Hardware Interrupt

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 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 interrupts are 2, 3, 6-9, 14-17, 22-25, 29-32, 37, 38, 41-44, 52.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. 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 model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Simulink model task priority values are required to generate the proper rate transition code (refer to Rate Transitions and Asynchronous Blocks in the Simulink Coder documentation). The task priority values are also required to ensure 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 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 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 `15` in **Interrupt numbers** is not preemptible and the priority `42` 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.

Purpose

Receive UDP packet

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

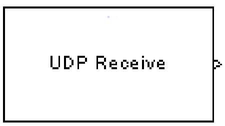
Embedded Coder/ Embedded Targets/ Operating Systems/ VxWorks

Simulink Coder/ Desktop Targets/ Host Communication

Windows (windowslib)

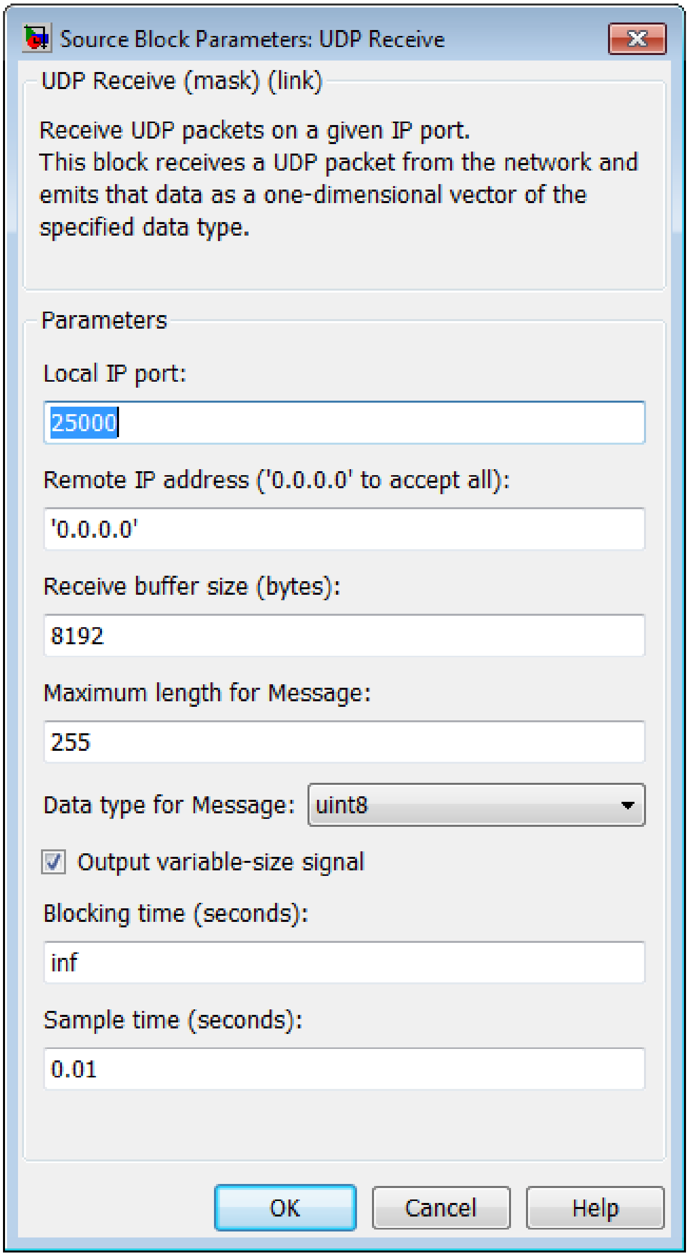
Note If your target system uses Linux or Windows, get the UDP block from the appropriate library, `linuxlib` or `windowslib`.

Description



UDP Receive

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block output, emits the contents of a single UDP packet as a data vector.



Dialog

Local IP port

Specify the IP port number upon to receive UDP packets. This value defaults to 25000. The value can range 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Remote IP address ('0.0.0.0' to accept all)
Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from any other address. To accept packets from any IP address, enter `'0.0.0.0'`. This value defaults to `'0.0.0.0'`.

Receive buffer size (bytes)
Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to `8192`.

Maximum length for Message
Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of any UDP packet. The system truncates data that exceeds this length. This value defaults to `255`.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message
Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Output variable-size signal
If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.
- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length
Set the data type of the Length output. This option defaults to `double`.

Blocking time (seconds)
For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note This parameter appears only in the UDP Receive block.

Sample time (seconds)
Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a

large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Output port width

Specify the width of packets the block accepts. When you design the transmit end of the UDP communication channel, you decide the packet width. Set this option to a value as large or larger than any packet you expect to receive.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

UDP receive buffer size (bytes)

Specify the size of the buffer to which the system stores UDP packets. The default size is 8192 bytes. Make the buffer large enough to store UDP packets that come in while your process reads a packet from the buffer or performs other tasks. Specifying the buffer size prevents the receive buffer from overflowing.

Note This parameter appears only in a deprecated version of the UDP Receive block. Replace the deprecated UDP Receive block with a current UDP Receive block.

See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Send

UDP Send

Purpose

Send UDP message

Library

Embedded Coder/ Embedded Targets/ Host Communication

Embedded Coder/ Embedded Targets/ Operating Systems/ Embedded Linux

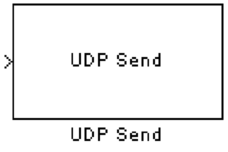
Embedded Coder/ Embedded Targets/ Operating Systems/ VxWorks

Simulink Coder/ Desktop Targets/ Host Communication

Windows (windowslib)

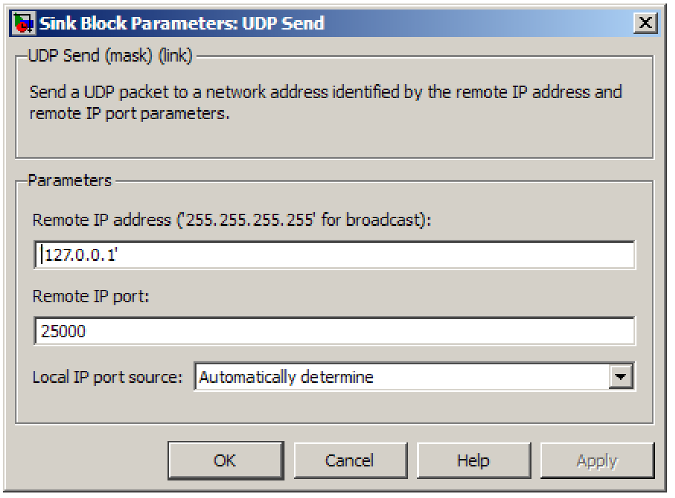
Note If your target system uses Linux or Windows, get the UDP block from the appropriate library, `linuxlib` or `windowslib`.

Description



The UDP Send block transmits an input vector as a UDP message over an IP network port.

Dialog Box



IP address (`'255.255.255.255'` for broadcast)
Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, `'255.255.255.255'`.

Remote IP port
Specify the port to which the block sends the message. The value defaults to 25000, but the values range from 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Local IP port source

To let the system automatically assign the port number, select `Assign automatically`. To specify the IP port number using the **Local IP port** parameter, select `Specify`.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Sample time

Sample time tells the block how long to wait before polling for new messages.

Note This parameter only appears in a deprecated version of the UDP Send block. Replace the deprecated UDP Send block with a current UDP Send block.

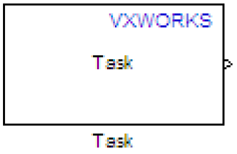
See Also

Byte Pack, Byte Reversal, Byte Unpack, UDP Receive

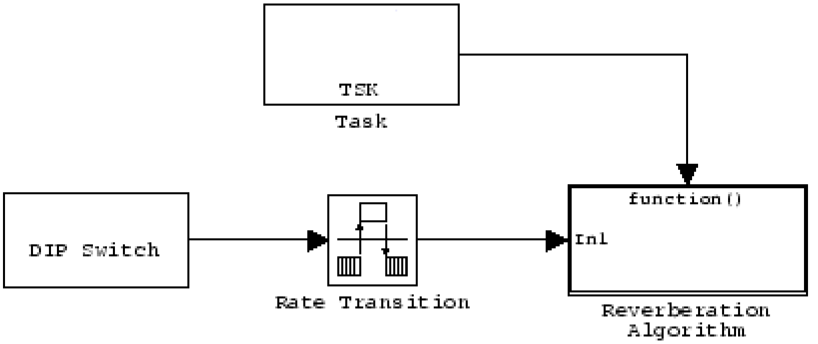
Purpose Spawn task function as separate VxWorks thread

Library Embedded Coder/ Embedded Targets/ Operating Systems/ VxWorks

Description

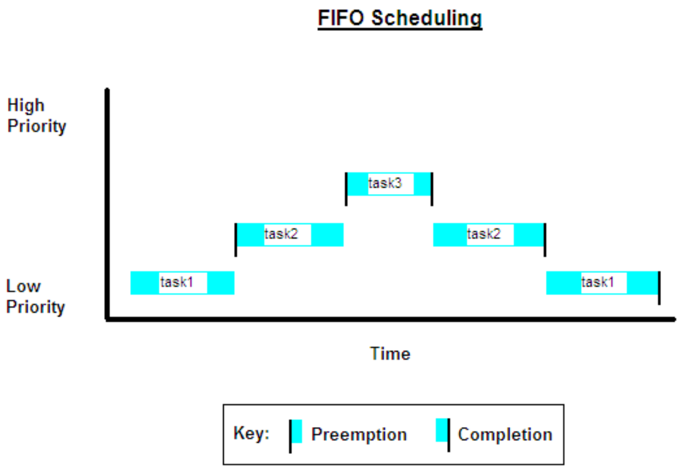


Use this block to create a task function that spawns as a separate VxWorks thread. The task function runs the code of the downstream function-call subsystem.For example:

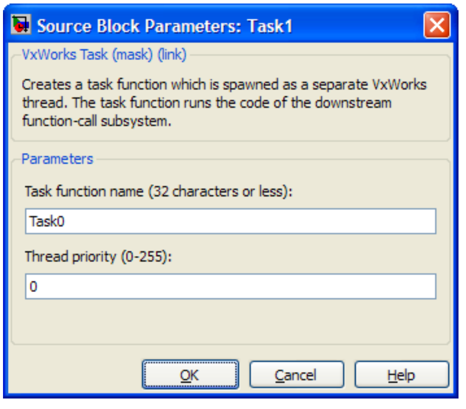


The VxWorks Task block uses a First In, First Out (FIFO) scheduling algorithm, which executes real-time processes without time slicing. With FIFO scheduling, a higher-priority process preempts a lower-priority process. While the higher-priority process runs, the lower-priority process remains at the top of the list for its priority. When the scheduler blocks all 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.

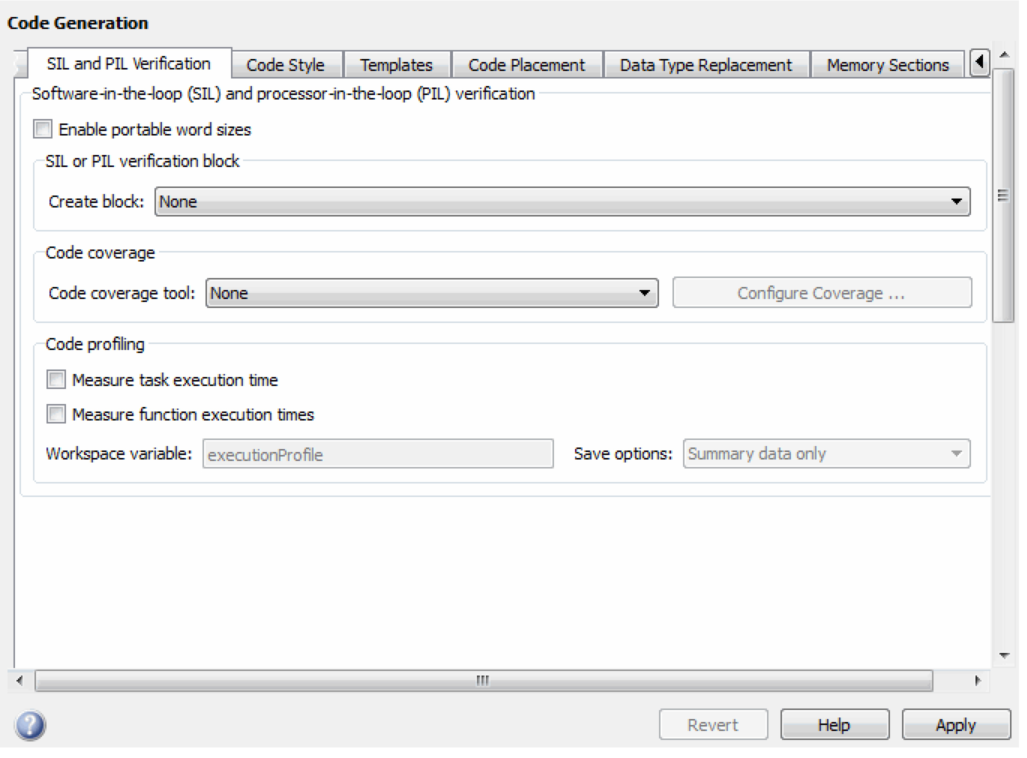
Thread priority (0 to 255)

Set the priority for the thread, from 0 to 255 (low-to-high). Higher-priority tasks can preempt lower-priority tasks.

See Also

Configuration Parameters

Code Generation Pane: SIL and PIL Verification



In this section...
“Code Generation: SIL and PIL Verification Tab Overview” on page 6-3
“Enable portable word sizes” on page 6-4
“Create block” on page 6-6
“Code coverage tool” on page 6-8
“Measure task execution time” on page 6-9
“Measure function execution times” on page 6-11
“Workspace variable” on page 6-13
“Save options” on page 6-15

Code Generation: SIL and PIL Verification Tab Overview

Create SIL block, configure word size portability, and configure code coverage for SIL testing

Configuration

This tab appears only if you specify an ERT–based system target file.

See Also

"SIL and PIL Simulation"

Enable portable word sizes

Specify whether to allow portability across host and target processors that support different word sizes.

You can enable portable word sizes to support SIL testing of your generated code. 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
: Generates conditional processing macros that support compilation of generated code on a processor that supports a different word size than the target processor on which production code is intended to run (for example, a 32-bit host and a 16-bit target. This allows you to use the same generated code for both software-in-the-loop (SIL) testing on the host platform and production deployment on the target platform.

☐ Off
: Does not generate portable code.

Dependencies

When you use this parameter, you should set **Emulation hardware** on the **Hardware Implementation** pane to `None`.

Command-Line Information

Parameter: `PortableWordSizes`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	No impact

See Also

- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes
- Tips for Optimizing the Generated Code
- “Verification”

Create block

Generate a SIL or PIL block

Settings

Default: `None`

`None`
: No SIL or PIL block generated.

`SIL`
: Create a SIL block with an S-function to represent the model or subsystem. The coder generates an inlined C or C++ MEX S-function wrapper that calls existing handwritten code or code previously generated by the code generation software from within the Simulink product. S-function wrappers provide a standard interface between the Simulink product and externally written code, allowing you to integrate your code into a model with minimal modification.

When you select this option, the software:

1. Generates the S-function wrapper file `model_sf.c` (or `.cpp`) and places it in the build directory.
2. Builds the MEX-file `model_sf.mexext` and places it in your working directory.
3. Creates and opens an untitled model with a SIL block containing the S-function.

`PIL`
: Create a PIL block that contains cross-compiled object code for a target processor or equivalent instruction set simulator. When you select this option, the software creates and opens an untitled model with a PIL block. With this block, you can verify the behavior of object code generated from subsystem or top-model components.

Use Target Connectivity API to control the way code compiles and executes in the target environment.

Command-Line Information

Parameter: `CreateSILPILBlock`
Type: string
Value: `'None'` | `'SIL'` | `'PIL'`
Default: `'None'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Automatic S-Function Wrapper Generation
- Techniques for Exporting Function-Call Subsystems
- Validating ERT Production Code on the MATLAB Host Computer Using Portable Word Sizes
- "SIL and PIL Simulation"

Code coverage tool

Specify a code coverage tool

Settings

Default: `None`

`None`
: No code coverage tool specified

`BullseyeCoverage`
: Specifies the BullseyeCoverage™ tool from Bullseye Testing Technology™

Dependencies

You cannot specify this parameter if **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.

See Also

- "Using a Code Coverage Tool in SIL Simulation"
- "Configuring Code Coverage Programmatically"

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. Data obtained from instrumentation probes added to SIL or PIL test harness.

☐ Off
  No measurement 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.

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

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"
- "Analyzing 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
: No execution times collected for functions inside generated code

Dependencies

To use this parameter, you must also select the **Measure task execution time** check box and specify a workspace variable.

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

- "Configuring Code Execution Profiling"
- "Viewing Code Execution Reports"
- "Analyzing 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 `rtw.pil.ExecutionProfile` object. To view and analyse execution profiles, use methods from the `rtw.pil.ExecutionProfile` and `rtw.pil.ExecutionProfileSection` 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: any valid string
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	No impact

See Also

- "Configuring Code Execution Profiling"

- "Viewing Code Execution Reports"
- "Analyzing Code Execution Data"

Save options

Specify whether to save all 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 `rtw.pil.ExecutionProfile` 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 all 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 `rtw.pil.ExecutionProfile` `report` method.

`All measurement and analysis data`
: Save all code profiling measurement and analysis data to a `rtw.pil.ExecutionProfile` object in the base workspace. In addition to viewing the code execution report, this option allows you to analyze data using `rtw.pil.ExecutionProfile` and `rtw.pil.ExecutionProfileSection` 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

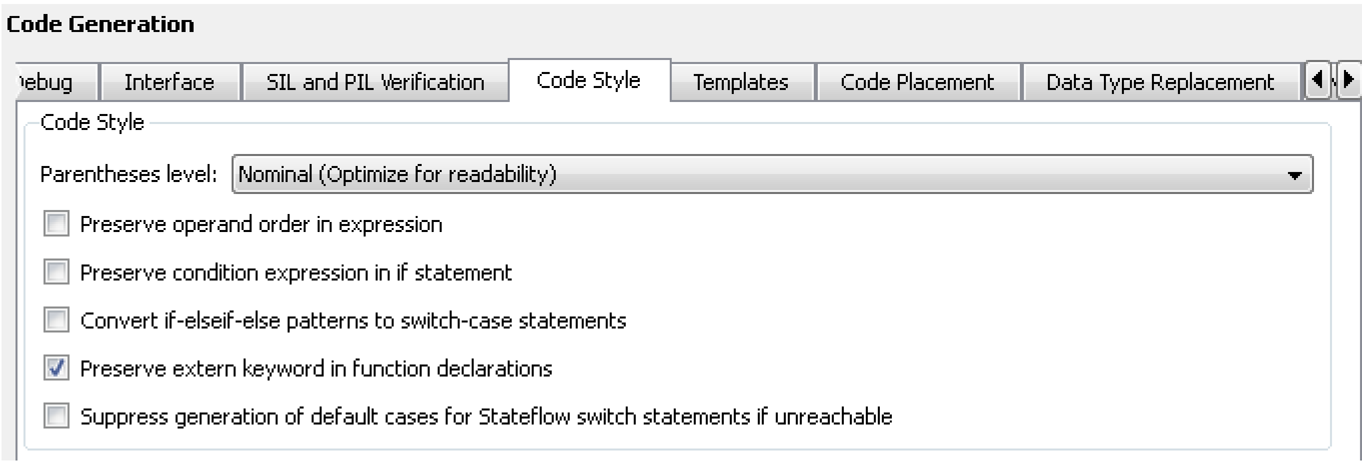
Parameter: `CodeProfilingSaveOptions`
Type: string
Value: `'SummaryOnly'` | `'AllData'`
Default: `'SummaryOnly'`

See Also

- "Configuring Code Execution Profiling"

- “Viewing Code Execution Reports”
- “Analyzing Code Execution Data”

Code Generation Pane: Code Style



In this section...

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

- “Controlling Code Style”
- “Code Generation Pane: Code Style” on page 6-17

Parentheses level

Specify parenthesization style for generated code.

Settings

Default: `Nominal (Optimize for readability)`

`Minimum (Rely on C/C++ operators for precedence)`

Inserts parentheses only where required by ANSI[6] C or C++, or needed to override default precedence. For example:

```
isZero = var == 0;
if (isZero == 1 && (value < 3.7 ||value > 9.27)) {
   /* code */
   }
```

`Nominal (Optimize for readability)`

Inserts parentheses in a way that compromises between readability and visual complexity. The exact definition can change between releases.

`Maximum (Specify precedence with parentheses)`

Includes parentheses everywhere needed to specify meaning without relying on operator precedence. Code generated with this setting conforms to MISRA®[7] requirements. For example:

```
isZero = (var == 0);
if ((isZero == 1) && ((value < 3.7) || (value > 9.27))) {
   /* code */
   }
```

Command-Line Information

Parameter: `ParenthesesLevel`
Type: string
Value: `'Minimum'` | `'Nominal'` | `'Maximum'`
Default: `'Nominal'`

6. ANSI® is a registered trademark of the American National Standards Institute, Inc.
7. MISRA® is a registered trademarks of MIRA Ltd, held on behalf of the MISRA® Consortium.

Recommended Settings

Application	Setting
Debugging	Nominal (Optimized for readability)
Traceability	Nominal (Optimized for readability)
Efficiency	Minimum (Rely on C/C++ operators for precedence)
Safety precaution	Maximum (Specify precedence with parentheses)

See Also

Controlling Parenthesization

Preserve operand order in expression

Specify whether to preserve order of operands in expressions.

Settings

Default: off

☑ On

Preserves the expression order specified in the model. Select this option to increase readability of the code or for code traceability purposes.

```
A*(B+C)
```

☐ Off

Optimizes efficiency of code for nonoptimized compilers by reordering commutable operands to make expressions left-recursive. For example:

```
(B+C)*A
```

Command-Line Information

Parameter: `PreserveExpressionOrder`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	On

Preserve condition expression in if statement

Specify whether to preserve empty primary condition expressions in `if` statements.

Settings

Default: off

☑ On

Preserves empty primary condition expressions in `if` statements, such as the following, to increase the readability of the code or for code traceability purposes.

```
if expression1
else
       statements2;
end
```

☐ Off

Optimizes empty primary condition expressions in `if` statements by negating them. For example, consider the following `if` statement:

```
if expression1
else
       statements2;
end
```

By default, the code generator negates this statement as follows:

```
if ~expression1
        statements2;
end
```

Command-Line Information

Parameter: `PreserveIfCondition`
Type: string
Value: `'on' | 'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	Off
Safety precaution	On

Convert if-elseif-else patterns to switch-case statements

Specify whether to generate code for `if-elseif-else` decision logic as `switch-case` statements.

This readability optimization works on a per-model basis and applies only to:

- Flow graphs in Stateflow charts
- 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

- "Enhancing Readability of Generated Code for Flow Graphs"
- "Enhancing Readability of Generated Code for MATLAB Function Blocks"
- "Controlling Code Style"

Preserve extern keyword in function declarations

Specify whether to include the `extern` keyword in function declarations in the generated code.

Note The `extern` keyword is optional for functions with external linkage. It is considered good programming practice to include the `extern` keyword in function declarations for code readability.

Settings

Default: on

☑ On

Include the `extern` keyword in function declarations in the generated code. For example, the generated code for the demo model `rtwdemo_hyperlinks` contains the following function declarations in `rtwdemo_hyperlinks.h`:

```
/* Model entry point functions */
extern void rtwdemo_hyperlinks_initialize(void);
extern void rtwdemo_hyperlinks_step(void);
```

The `extern` keyword explicitly indicates that the function has external linkage. The function definitions in this example are in the generated file `rtwdemo_hyperlinks.c`.

☐ Off

Remove the `extern` keyword from function declarations in the generated code.

Command-Line Information

Parameter: `PreserveExternInFcnDecls`
Type: string
Value: `'on' | 'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information on code style options, see "Code Generation Pane: Code Style" on page 6-17.

Suppress generation of default cases for Stateflow switch statements if unreachable

Specify whether or not to generate default cases for switch-case statements in the code for Stateflow charts. This optimization works on a per-model basis and applies to the code generated for a state that has multiple substates. For a list of the state functions in the generated code, see “Controlling Inlining of 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

Always generate a default case whether or not it is reachable. This setting supports MISRA C® compliance and provides a fallback 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 appropriate substate as active.

Command-Line Information

Parameter: `SuppressUnreachableDefaultCases`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

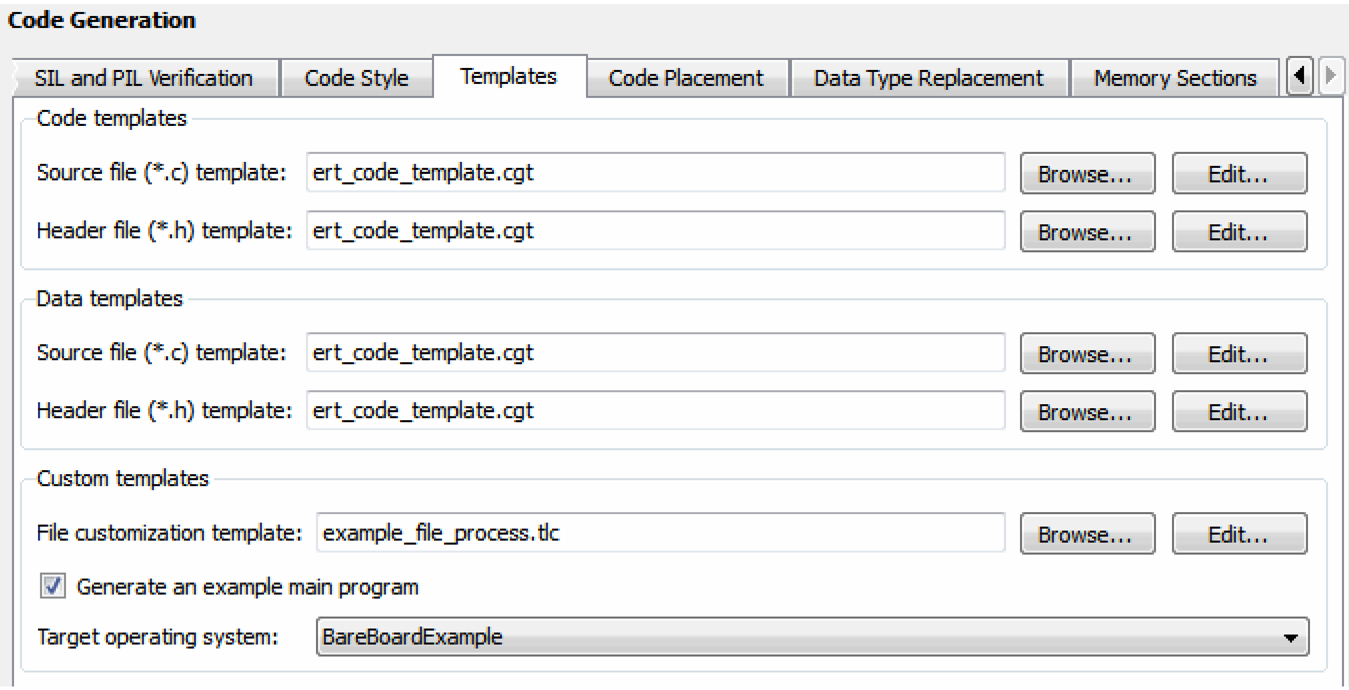
Recommended Settings

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 6-17.

Code Generation Pane: Templates



In this section...

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 6-30

Code templates: Source file (*.c) template

Specify the code generation template (CGT) file to use when generating a source code file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated source code files (`.c` or `.cpp`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTSrcFileBannerTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

Code templates: Header file (*.h) template

Specify the code generation template (CGT) file to use when generating a code header file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated header files (`.h`).

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTHdrFileBannerTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

Data templates: Source file (*.c) template

Specify the code generation template (CGT) file to use when generating a data source file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data source files (`.c` or `.cpp`) that contain definitions of variables of global scope.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataSrcFileTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

Data templates: Header file (*.h) template

Specify the code generation template (CGT) file to use when generating a data header file.

Settings

Default: `ert_code_template.cgt`

You can use a CGT file to define the top-level organization and formatting of generated data header files (`.h`) that contain declarations of variables of global scope.

Note The CGT file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTDataHdrFileTemplate`
Type: string
Value: any valid CGT file
Default: `'ert_code_template.cgt'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

File customization template

Specify the custom file processing (CFP) template file to use when generating code.

Settings

Default: `ert_code_template.tlc`

You can use a CFP template file to customize generated code. A CFP template file is a TLC file that organizes types of code (for example, includes, `typedefs`, and functions) into sections. The primary purpose of a CFP template is to assemble code to be generated into buffers, and to call a code template API to emit the buffered code into specified sections of generated source and header files. The CFP template file must be located on the MATLAB path.

Command-Line Information

Parameter: `ERTCustomFileTemplate`
Type: string
Value: any valid TLC file
Default: `'example_file_process.tlc'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

Generate an example main program

Control whether to generate an example main program for a model.

Settings

Default: on

☑ On

Generates an example main program, `ert_main.c` (or `.cpp`). The file includes:

- The `main()` function for the generated program
- Task scheduling code that determines how and when block computations execute on each time step of the model

The operation of the main program and the scheduling algorithm employed depend primarily on whether your model is single-rate or multirate, and also on your model's solver mode (`SingleTasking` or `MultiTasking`).

☐ Off

Provides a static version of the file `ert_main.c` as a basis for custom modifications (*`matlabroot`*`/rtw/c/ert/ert_main.c`). You can use this file as a template for developing embedded applications.

Tips

- After you generate and customize the main program, disable this option to prevent regenerating the main module and overwriting your customized version.
- You can use a custom file processing (CFP) template file to override normal main program generation, and generate a main program module customized for your target environment.
- If you disable this option, the coder generates slightly different rate grouping code to maintain compatibility with an older static `ert_main.c` module.

Dependencies

- This parameter enables **Target operating system**.
- You must enable this parameter and select `VxWorksExample` for **Target operating system** if you use VxWorks®[8] library blocks.

Command-Line Information

Parameter: `GenerateSampleERTMain`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Generating a Standalone Program"
- Static Main Program Module
- Custom File Processing

8. VxWorks® is a registered trademark of Wind River® Systems, Inc.

Target operating system

Specify a target operating system to use when generating model-specific example main program module.

Settings

Default: `BareBoardExample`

`BareBoardExample`
: Generates a bareboard main program designed to run under control of a real-time clock, without a real-time operating system.

`VxWorksExample`
: Generates a fully commented example showing how to deploy the code under the VxWorks real-time operating system.

Dependencies

- This parameter is enabled by **Generate an example main program**.
- This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `TargetOS`
Type: string
Value: `'BareBoardExample'` | `'VxWorksExample'`
Default: `'BareBoardExample'`

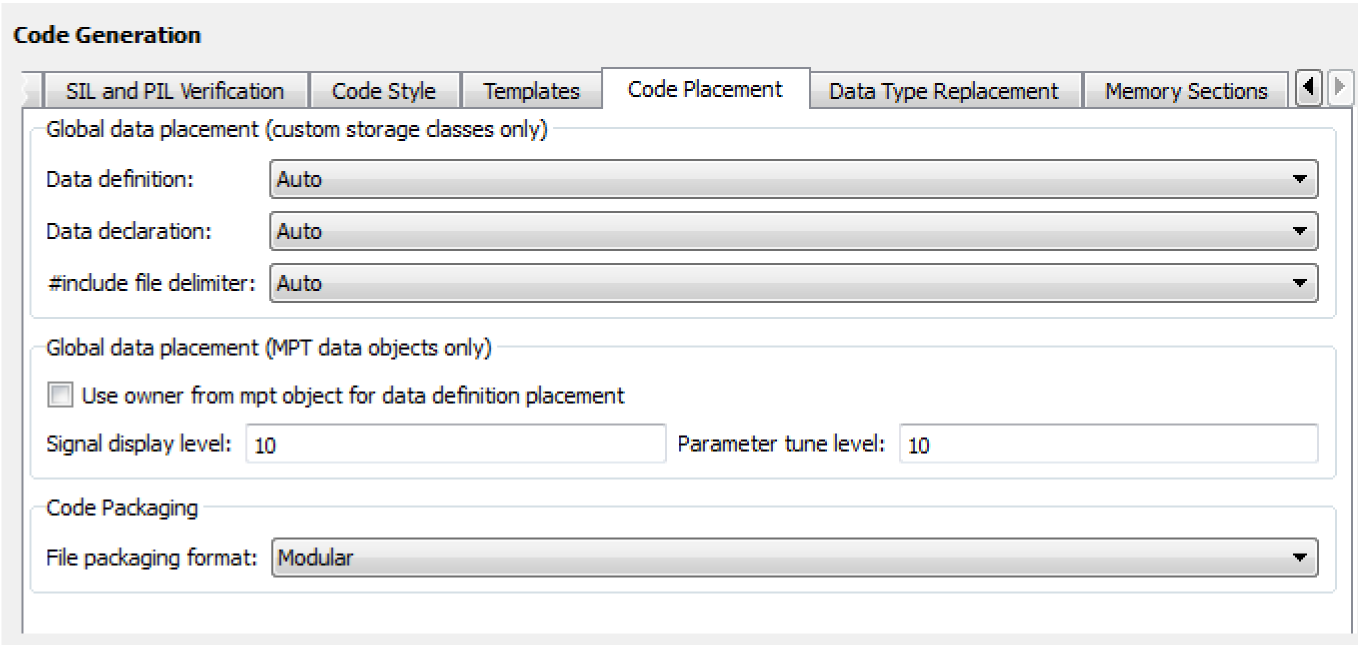
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Generating a Standalone Program"
- Static Main Program Module
- Custom File Processing

Code Generation Pane: Code Placement



In this section...

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

- "Data, Function, and File Definition"
- "Code Generation Pane: Code Placement" on page 6-41

Data definition

Specify where to place definitions of global variables.

Settings

Default: `Auto`

`Auto`
: Lets the code generator determine where the definitions should be located.

`Data defined in source file`
: Places definitions in `.c` source files where functions are located. The code generator places the definitions in one or more function `.c` files, depending on the number of function source files and the file partitioning previously selected in the Simulink model.

`Data defined in a single separate source file`
: Places definitions in the source file specified in the **Data definition filename** field. The code generator organizes and formats the definitions based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Templates** pane.

Dependencies

- This parameter applies to data with custom storage classes only.
- This parameter enables **Data definition filename**.

Command-Line Information

Parameter: `GlobalDataDefinition`
Type: string
Value: `'Auto' | 'InSourceFile' | 'InSeparateSourceFile'`
Default: `'Auto'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid value
Efficiency	No impact
Safety precaution	No impact

See Also

- "Overview of Data Placement"
- "Managing Placement of Data Definitions and Declarations"
- "Data Placement Rules and Effects"

Data definition filename

Specify the name of the file that is to contain data definitions.

Settings

Default: `global.c` or `global.cpp`

The code generator organizes and formats the data definitions in the specified file based on the data source template specified by the **Source file (*.c) template** parameter in the data section of the **Code Generation** pane: **Templates** tab.

If you specify C++ as the target language, omit the `.cpp` extension. The code generator will generate the correct file and add the extension `.cpp`.

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: any valid file
Default: `'global.c'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid file

Application	Setting
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	Any valid value
Efficiency	No impact
Safety precaution	No impact

See Also

- "Overview of Data Placement"
- "Managing Placement of Data Definitions and Declarations"
- "Data Placement Rules and Effects"

Data declaration filename

Specify the name of the file that is to contain data declarations.

Settings

Default: `global.h`

The code generator organizes and formats the data declarations in the specified file based on the data header template specified by the **Header file (*.h) template** parameter in the data section of the **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: any valid file
Default: `'global.h'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid file
Efficiency	No impact
Safety precaution	No impact

See Also

- Selecting and Defining Templates
- Custom File Processing

Use owner from mpt object for data definition placement

Specify whether the model uses or ignores the ownership setting of an mpt data object for data definition in code generation.

Settings

Default: off

☑ On

Uses the ownership setting of the mpt data object for data definition. This value corresponds to the `SameAsModel` value of the `ModuleNamingRule` parameter.

☐ Off

Ignores the ownership setting of the mpt 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: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any 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	Any valid value
Efficiency	No impact
Safety precaution	No impact

Signal display level

Specify the persistence level for all MPT signal data objects.

Settings

Default: 10

Specify an integer value indicating the persistence level for all MPT signal data objects. This value indicates the level at which to declare signal data objects as global data in the generated code. The persistence level allows you to make intermediate variables global during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value that you can specify for a specific MPT signal data object in the Model Explorer signal properties dialog.

Dependency

This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `SignalDisplayLevel`
Type: integer
Value: any valid integer
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid integer
Efficiency	No impact
Safety precaution	No impact

See Also

Selecting Persistence Level for Signals and Parameters

Parameter tune level

Specify the persistence level for all MPT parameter data objects.

Settings

Default: 10

Specify an integer value indicating the persistence level for all MPT parameter data objects. This value indicates the level at which to declare parameter data objects as tunable global data in the generated code. The persistence level allows you to make intermediate variables global and tunable during initial development so you can remove them during later stages of development to gain efficiency.

This parameter is related to the **Persistence level** value you that can specify for a specific MPT parameter data object in the Model Explorer parameter properties dialog.

Dependency

This parameter must be the same for top-level and referenced models.

Command-Line Information

Parameter: `ParamTuneLevel`
Type: integer
Value: any valid integer
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid integer
Efficiency	No impact
Safety precaution	No impact

See Also

Selecting Persistence Level for Signals and Parameters

File packaging format

Specify whether code generation modularizes the code components into many files or compacts the generated code into a few files. You can specify a different file packaging format for each referenced model.

Settings

Default: `Modular`

`Modular`

- Outputs *`model`*`_data.c`, *`model`*`_private.h`, and *`model`*`_types.h`, in addition to generating *`model`*`.c` and *`model`*`.h`. For the contents of these files, see the table in "Generated Code Modules".
- Supports generating separate source files for subsystems. For more information on generating code for subsystems, see "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'

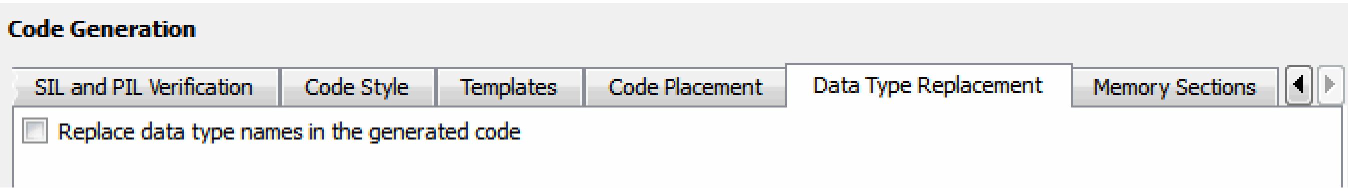
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

- "Customizing Generated Code Modules"
- "Generating Code Modules"
- "Customizing Post-Code-Generation Build Processing"

Code Generation Pane: Data Type Replacement



In this section...

Code Generation: Data Type Replacement Tab Overview

Replace built-in data type names with user-defined replacement data type names in the generated code for your model.

Configuration

This tab appears only if you specify an ERT based System target file.

If your application requires you to replace built-in data type names with user-defined replacement data type names in the generated code:

1. Select **Replace data type names in the generated code**.
2. Selectively specify replacement data type names to use for built-in Simulink data types in the **Replacement Name** fields.

See Also

- "Data Type Replacement"
- "Code Generation Pane: Data Type Replacement" on page 6-58

Replace data type names in the generated code

Specify whether to replace built-in data type names with user-defined data type names in generated code.

Settings

Default: off

☑ On

Displays the **Data type names** table. The table provides a way for you to replace the names of built-in data types used in generated code. This mechanism can be particularly useful for generating code that adheres to application or site data type naming standards.

You can choose to specify new data type names for some or all Simulink built-in data types listed in the table. For each replacement data type name that you specify:

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, and `uint8`, the `BaseType` of the replacement data type must match the built-in data type.
- For `boolean`, the `BaseType` of the replacement data type must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.
- For `int`, `uint`, and `char`, the size of the replacement data type must match the size displayed for **Number of bits: int** or **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

☐ Off

Uses Simulink Coder names for built-in Simulink data types in generated code.

Dependencies

This parameter enables:

double Replacement Name
single Replacement Name
int32 Replacement Name
int16 Replacement Name
int8 Replacement Name
uint32 Replacement Name
uint16 Replacement Name
uint8 Replacement Name
boolean Replacement Name
int Replacement Name
uint Replacement Name
char Replacement Name

Command-Line Information

Parameter: `EnableUserReplacementTypes`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

"Data Type Replacement"

Replacement Name: double

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.double`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: single

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.single`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: int32

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int32`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: int16

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types .

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int16`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: int8

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int8`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: uint32

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint32`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: uint16

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint16`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: uint8

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The `BaseType` of the replacement data type must match the built-in data type.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint8`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and `BaseType` of the replacement data type must match the built-in data type
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: boolean

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.boolean`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	`''`

See Also

- “Replacing `boolean` with an Integer Data Type”
- “Data Type Replacement”

Replacement Name: int

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: ''

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.int`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and the size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: ''

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid value
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: uint

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.uint`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and the size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Replacement Name: char

Specify names to use for built-in Simulink data types in generated code.

Settings

Default: `''`

Specify strings that the code generator is to use as names for built-in Simulink data types.

- The name must match the name of a `Simulink.AliasType` object that exists in the base workspace.
- The `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.
- The size of the replacement data type must match the size displayed for on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

Dependency

This parameter is enabled by **Replace data type names in the generated code**.

Command-Line Information

Parameter: `ReplacementTypes, replacementName.char`
Type: string
Value: name of a `Simulink.AliasType` object that exists in the base workspace; `BaseType` property of object must be consistent with the built-in data type it replaces and the size of the replacement data type must match the size displayed on the **Hardware Implementation** pane of the Configuration Parameters dialog box
Default: `''`

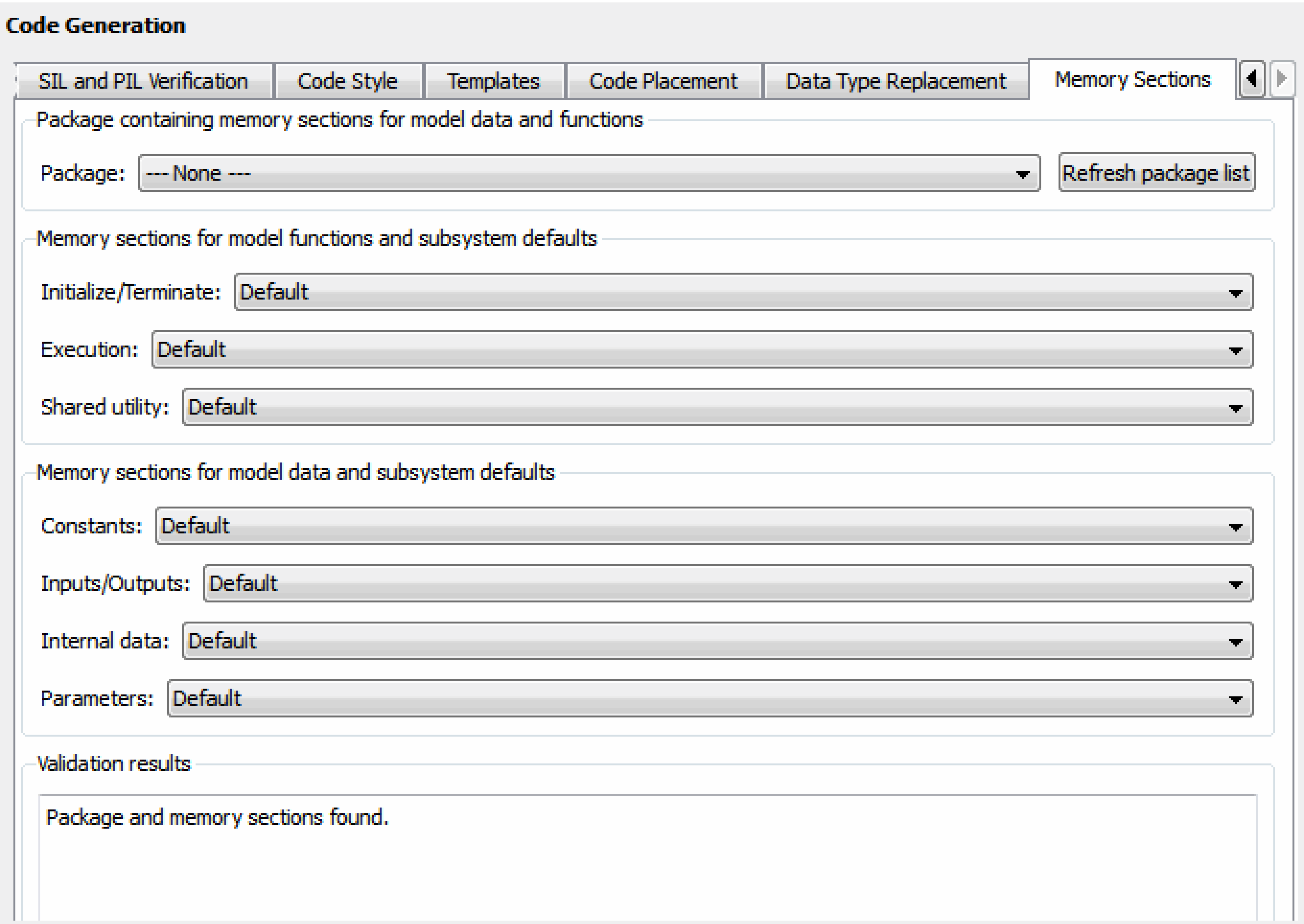
Recommended Settings

Application	Setting
Debugging	No impact
Traceability	Any valid string
Efficiency	No impact
Safety precaution	''

See Also

"Data Type Replacement"

Code Generation Pane: Memory Sections



In this section...

In this section...

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 6-86

Package

Specify a package that contains memory sections you want to apply to model-level functions and internal data.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `---None---`

`---None---`
: Suppresses memory sections.

`Simulink`
: Applies the built-in Simulink package.

`mpt`
: Applies the built-in mpt package.

Tip

If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

Command-Line Information

Parameter: `MemSecPackage`
Type: string
Value: `'--- None ---'` | `'Simulink'` | `'mpt'`
Default: `'--- None ---'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

"Memory Sections"

Refresh package list

Add user-defined packages that are on the search path to list of packages displayed by **Packages**.

Tip

If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

See Also

"Memory Sections"

Initialize/Terminate

Specify whether to apply a memory section to Initialize/Start and Terminate functions.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for Initialize, Start, and Terminate functions.

`memory-section-name`
: Applies a memory section to Initialize, Start, and Terminate functions.

Command-Line Information

Parameter: `MemSecFuncInitTerm`
Type: string
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Memory Sections"

Execution

Specify whether to apply a memory section to execution functions.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for Step, Run-time initialization, Derivative, Enable, and Disable functions.

`memory-section-name`
: Applies a memory section to Step, Run-time initialization, Derivative, Enable, and Disable functions.

Command-Line Information

Parameter: `MemSecFuncExecute`
Type: string
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Memory Sections"

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 all 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 all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
  Suppresses the use of a memory section for constants.

`memory-section-name`
  Applies a memory section to constants.

This parameter applies to:

Data Definition	Data Purpose
*model*_cP	Constant parameters
*model*_cB	Constant block I/O
*model*_Z	Zero representation

Command-Line Information

Parameter: `MemSecDataConstants`
Type: string
Value: `'Default' | 'MemConst' | 'MemVolatile' | 'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

"Memory Sections"

Inputs/Outputs

Specify whether to apply a memory section to root input and output.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for root-level input and output.

`memory-section-name`
: Applies a memory section for root-level input and output.

This parameter applies to:

Data Definition	Data Purpose
`model``_U`	Root-level input
`model``_Y`	Root-level output

Command-Line Information

Parameter: `MemSecDataIO`
Type: string
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

"Memory Sections"

Internal data

Specify whether to apply a memory section to internal data.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppresses the use of a memory section for internal data.

`memory-section-name`
: Applies a memory section for internal data.

This parameter applies to:

Data Definition	Data Purpose
`model``_B`	Block I/O
`model``_D`	DWork vectors
`model``_M`	Run-time model
`model``_Zero`	Zero-crossings

Command-Line Information

Parameter: `MemSecDataInternal`
Type: string
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

"Memory Sections"

Parameters

Specify whether to apply a memory section to parameters.

Settings

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems except atomic subsystems that contain overriding memory section specifications.

Default: `Default`

`Default`
: Suppress the use of a memory section for parameters.

`memory-section-name`
: Apply memory section for parameters.

This parameter applies to:

Data Definition	Data Purpose
`model``_P`	Parameters

Command-Line Information

Parameter: `MemSecDataParameters`
Type: string
Value: `'Default'` | `'MemConst'` | `'MemVolatile'` | `'MemConstVolatile'`
Default: `'Default'`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Memory Sections

Validation results

Display the results of memory section validation.

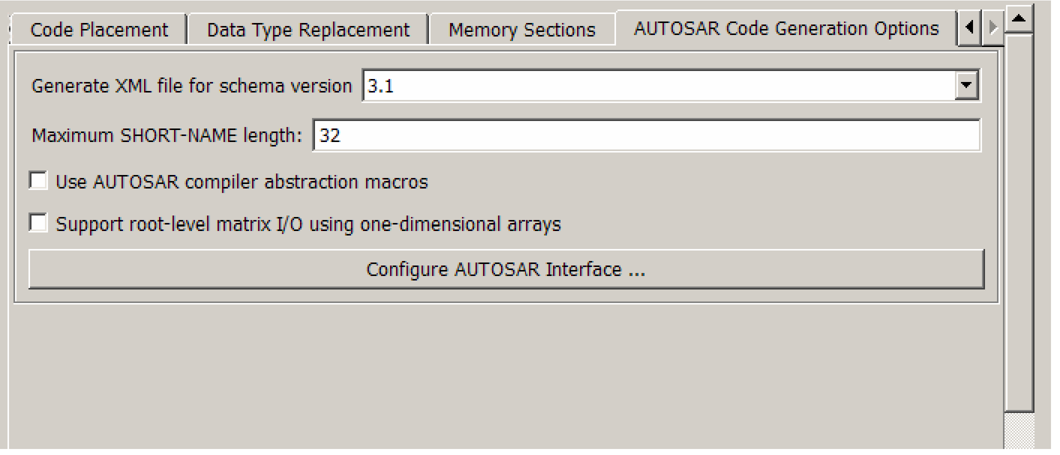
Settings

The 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



In this section...

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

Click the **Configure AUTOSAR Interface** button to open a dialog box where you can configure all other AUTOSAR options.

See Also

- "Generating Code for AUTOSAR Software Components"
- "AUTOSAR Configuration" on page 1-4
- "AUTOSAR" on page 1-3
- "Code Generation Pane: AUTOSAR Code Generation Options" on page 6-104

Generate XML file from schema version

Select the AUTOSAR schema version to use when generating XML files.

Settings

Default: `3.1`

`3.1`
: Use schema version 3.1

`3.0`
: Use schema version 3.0

`2.1`
: Use schema version 2.1

`2.0`
: Use schema version 2.0

Tip

Click the **Configure AUTOSAR Interface** button to open a dialog box where you can configure all other AUTOSAR options.

Command-Line Information

Parameter: `AutosarSchemaVersion`
Type: string
Value: `'3.1'` | `'3.0'` | `'2.1'` | `'2.0'`
Default: `'3.1'`

See Also

"Generating Code for AUTOSAR Software Components"

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: any integer less or equal to 128
Default: `32`

See Also

"Specifying 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

"Configuring 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"

Configure AUTOSAR Interface

Opens the Model Interface dialog box where you can configure all other AUTOSAR options.

Dependencies

This parameter is disabled if you are using Configuration Set Reference.

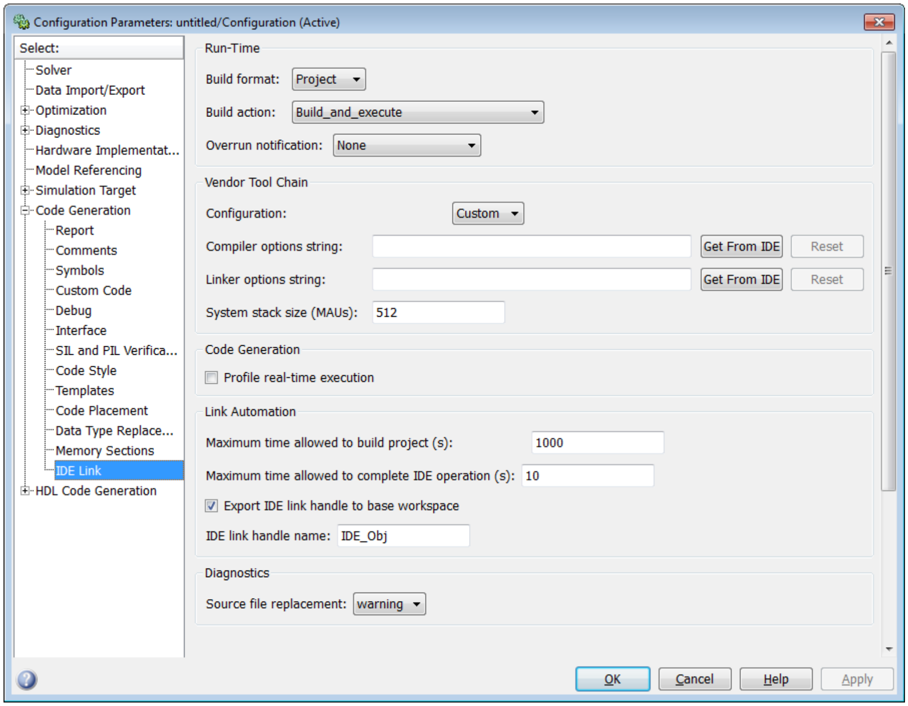
Command-Line Information

Parameter:`autosar_gui_launch`
Type: String
Value: *`subsystemName`*
Default: `No default`

See Also

- "Using the Configure AUTOSAR Interface Dialog Box"
- "Generating Code for AUTOSAR Software Components"

Code Generation Pane: IDE Link



In this section...

In this section...

Overview

Use this pane to configure the following parameters:

- Run-Time: set the build format to an IDE project or makefile, choose whether to build and execute the project, or create a PIL project.
- Vendor Tool Chain: set compiler and linker options.
- Code Generation: set options for profiling real-time execution.
- Link Automation: Set the maximum time to build projects and complete IDE operations. Set a default name for the IDE link handle.
- Diagnostics: Select the type of message to generate when the software replaces source files.

Build format

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: `Project`

`Project`
    Builds your model as an IDE project.

`Makefile`
    Creates a makefile and uses it to build your model.

Dependencies

Selecting `Makefile` removes the following parameters:

- **Code Generation**
  - **Profile real-time execution**
  - **Profile by**
  - **Number of profiling samples to collect**
- **Link Automation**
  - **Maximum time allowed to build project (s)**
  - **Maximum time allowed to complete IDE operation (s)**
  - **Export IDE link handle to base workspace**
  - **IDE link handle name**

Command-Line Information

Parameter: `buildFormat`
Type: string
Value: `Project` | `Makefile`
Default: `Build_and_execute`

Recommended Settings

Application	Setting
Debugging	`Project`
Traceability	`Project`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Build action

Defines how Simulink Coder software responds when you press Ctrl+B to build your model.

Settings

Default: `Build_and_execute`

If you set **Build format** to `Project`, select one of the following options:

`Build_and_execute`
: Builds your model, generates code from the model, and then compiles and links the code. After the software links your compiled code, the build process downloads and runs the executable on the processor.

`Create_project`
: Directs Simulink Coder software to create a new project in the IDE. The command line equivalent for this setting is `Create`.

`Archive_library`
: Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

`Build`
: Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

`Create_processor_in_the_loop_project`
: Directs the Simulink Coder code generation process to create PIL algorithm object code as part of the project build.

If you set **Build format** to `Makefile`, select one of the following options:

`Create_makefile`
: Creates a makefile. For example, ".mk". The command line equivalent for this setting is `Create`.

`Archive_library`
: Creates a makefile and an archive library. For example, ".a" or ".lib".

`Build`
: Creates a makefile and an executable. For example, ".exe".

`Build_and_execute`
  Creates a makefile and an executable. Then it evaluates the execute instruction under the **Execute** tab in the current XMakefile configuration.

Dependencies

Selecting `Archive_library` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Overrun notification**
- **Function name**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Get from IDE**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

Command-Line Information

Parameter: buildAction
Type: string
Value: Build | Build_and_execute | Create | Archive_library | Create_processor_in_the_loop_project
Default: Build_and_execute

Recommended Settings

Application	Setting
Debugging	Build_and_execute
Traceability	Archive_library
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

Overrun notification

Specifies how your program responds to overrun conditions during execution.

Settings

Default: None

None
: Your program does not notify you when it encounters an overrun condition.

Print_message
: Your program prints a message to standard output when it encounters an overrun condition.

Call_custom_function
: When your program encounters an overrun condition, it executes a function that you specify in **Function name**.

Tips

- The definition of the standard output depends on your configuration.

Dependencies

Selecting Call_custom_function enables the **Function name** parameter.

Setting this parameter to Call_custom_function enables the **Function name** parameter.

Command-Line Information

Parameter: overrunNotificationMethod
Type: string
Value: None | Print_message | Call_custom_function
Default: None

Recommended Settings

Application	Setting
Debugging	`Print_message` or `Call_custom_function`
Traceability	`Print_message`
Efficiency	`None`
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Function name

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

Settings

No Default

Dependencies

This parameter is enabled by setting **Overrun notification** to `Call_custom_function`.

Command-Line Information

Parameter: `overrunNotificationFcn`
Type: string
Value: no default
Default: no default

Recommended Settings

Application	Setting
Debugging	String
Traceability	String
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Configuration

Sets the Configuration for building your project from the model.

Settings

Default: `Custom`

`Custom`
: Lets the user apply a specialized combination of build and optimization settings.

  `Custom` applies the same settings as the Release project configuration in IDE, except:

  - The compiler options do not use any optimizations.
  - The memory configuration specifies a memory model that uses `Far Aggregate` for data and `Far` for functions.

`Debug`
: Applies the `Debug` Configuration defined by the IDE to the generated project and code.

`Release`
: Applies the `Release` project configuration defined by the IDE to the generated project and code.

Dependencies

- Selecting `Custom` disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting `Release` sets the **Compiler options string** to the settings defined by the IDE.
- Selecting `Debug` sets the **Compiler options string** to the settings defined by the IDE.

.

Command-Line Information

Parameter: `projectOptions`

Type: string
Value: `Custom` | `Debug` | `Release`
Default: `Custom`

Recommended Settings

Application	Setting
Debugging	`Custom` or `Debug`
Traceability	`Custom`, `Debug`, `Release`
Efficiency	`Release`
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the coder product does not set any optimization flags.

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 any optimizations.
- Setting **Configuration** to `Debug` applies the debug settings defined by the IDE.
- Setting **Configuration** to `Release` applies the release settings defined by the IDE.

Command-Line Information

Parameter: `compilerOptionsStr`
Type: string
Value: `Custom | Debug | Release`
Default: `Custom`

Recommended Settings

Application	Setting
Debugging	`Custom`
Traceability	`Custom`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the coder product does not set any linker options.

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: any valid linker option
Default: none

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory.

This parameter is used in all targets to allocate the stack size for the generated application. For example, with embedded processors that are not running an operating system, this parameter determines the total stack space that can be used for the application. For operating systems such as Linux or WindowsVxWorks, this value specifies the stack space allocated per thread.

This parameter also affects the “Maximum stack size (bytes)” parameter, located in the Optimization > Signals and Parameters pane.

Settings

Default: 8192

Minimum: 0

Maximum: Available memory

- Enter the stack size in minimum addressable units (MAUs). An MAU is typically 1 byte, but its size can vary by target processor.
- The software does not verify the value you entered is valid. Enter the correct value.

Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

When you set the **System target file** parameter on the **Code Generation** pane to `idelink_ert.tlc` or `idelink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization > Signals and Parameters** pane to `Inherit from target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of (**System stack size**/2) with 200,000 bytes and uses the smaller of the two values.

Command-Line Information

Parameter: `systemStackSize`
Type: int
Default: `8192`

Recommended Settings

Application	Setting
Debugging	int
Traceability	int
Efficiency	int
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

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++
- Green Hills MULTI
- IAR Embedded Workbench
- 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: IDE Link" topic.

Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

Settings

Default: Off

☑ On
   Adds instrumentation to the generated code to support execution profiling and generate the profiling report.

☐ Off
   Does not instrument the generated code to produce the profile report.

Dependencies

This parameter adds **Number of profiling samples to collect** and **Profile by**.

Selecting this parameter enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop project` removes this parameter.

Command-Line Information

Parameter: `ProfileGenCode`
Type: string
Value: `'on'` | `'off'`
Default: `'off'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics..

Profile by

Defines which execution profiling technique to use.

Settings

Default: `Task`

`Task`
: Profiles model execution by the tasks in the model.

`Atomic subsystem`
: Profiles model execution by the atomic subsystems in the model.

Dependencies

Selecting **Real-time execution profiling** enables this parameter.

Command-Line Information

Parameter: `profileBy`
Type: string
Value: `Task` | `Atomic subsystem`
Default: `Task`

Recommended Settings

Application	Setting
Debugging	`Task` or `Atomic subsystem`
Traceability	`Archive_library`
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

For more information about PIL and its uses, refer to the "Verifying Generated Code via Processor-in-the-Loop" topic.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics.

Number of profiling samples to collect

Specify the size of the buffer that holds the profiling samples. Enter a value that is 2 times the number of profiling samples.

Each task or subsystem execution instance represents one profiling sample. Each sample requires two memory locations, one for the start time and one for the end time. Consequently, the size of the buffer is twice the number of samples.

Sample collection begins with the start of code execution and ends when the buffer is full.

The profiling data is held in a statically sited buffer on the target processor.

Settings

Default: 100

Minimum: 2

Maximum: Buffer capacity

Tips

- Data collection stops when the buffer is full, but the application and processor continue running.
- Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

Dependencies

This parameter is enabled by **Profile real-time execution**.

Command-Line Information

Parameter:`ProfileNumSamples`
Type: int
Value: Positive integer
Default: `100`

Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

Settings

Default: 1000

Minimum: 1

Maximum: No limit

Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to complete IDE operation** timeout value.

Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

Command-Line Information

Parameter:`ideObjBuildTimeout`
Type: int
Value: Integer greater than 0
Default: `100`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Maximum time allowed to complete IDE operation (s)

specifies how long, in seconds, the software waits for IDE functions, such as `read` or `write`, to return completion messages.

Settings

Default: 10

Minimum: 1

Maximum: No limit

Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time.
- This timeout value does not depend on the global timeout value in a `IDE_Obj` object or the **Maximum time allowed to build project (s)** timeout value

Command-Line Information

Parameter:`'ideObjTimeout'`
Type: int
Value:
Default: `10`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Export IDE link handle to base workspace

Directs the software to export the `IDE_Obj` object to your MATLAB workspace.

Settings

Default: On

☑ On

Directs the build process to export the `IDE_Obj` object created to your MATLAB workspace. The new object appears in the workspace browser. Selecting this option enables the **IDE link handle name** option.

☐ Off

prevents the build process from exporting the `IDE_Obj` object to your MATLAB software workspace.

Dependency

Selecting **Profile real-time execution** enables **Export IDE link handle to base workspace** and makes it non-editable, since the coder software must create a handle.

Selecting **Export IDE link handle to base workspace** enables **IDE link handle name**.

Command-Line Information

Parameter: `exportIDEObj`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

Recommended Settings

Application	Setting
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the “Code Generation Pane: IDE Link” topic.

IDE link handle name

specifies the name of the IDE_Obj object that the build process creates.

Settings

Default: IDE_Obj

- Enter any valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the IDE_Obj object.
- The handle name is case sensitive.

Dependency

This parameter is enabled by **Export IDE link handle to base workspace**.

Command-Line Information

Parameter: ideObjName
Type: string
Value:
Default: IDE_Obj

Recommended Settings

Application	Setting
Debugging	Enter any valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Source file replacement

Selects the diagnostic action to take if the coder software detects conflicts that you are replacing source code with custom code.

Settings

Default: `warn`

`none`
: Does not generate warnings or errors when it finds conflicts.

`warning`
: Displays a warning.

`error`
: Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

Tips

- The build operation continues if you select `warning` and the software detects custom code replacement. You see warning messages as the build progresses.
- Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Select `none` when the replacement process is correct and you do not want to see multiple messages during your build.
- The messages apply to Simulink Coder **Custom Code** replacement options as well.

Command-Line Information

Parameter: `DiagnosticActions`
Type: string
Value: `none | warning | error`
Default: `warning`

Recommended Settings

Application	Setting
Debugging	error
Traceability	error
Efficiency	warning
Safety precaution	error

See Also

For more information, refer to the "Code Generation Pane: IDE Link" topic.

Parameter Reference

In this section...

Recommended Settings Summary

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" in the Simulink Coder documentation. 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` for 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

Mapping of Application Requirements to the Optimization Pane : General tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Remove internal data zero initialization	No impact	No impact	On (GUI) `off` (command line) (execution, ROM), No impact (RAM)	Off	Off
Optimize initialization code for model reference	No impact	No impact	On (execution, ROM), No impact (RAM)	No impact	On
Remove code that protects against division arithmetic exceptions	No impact	No impact	On	Off	Off

Mapping of Application Requirements to the Optimization Pane: Signals and Parameters tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Parameter structure	No impact	`Hierarchical`	`Non-Hierarchical`	No impact	`Hierarchical`
Pack Boolean data into bitfields	No impact	No Impact	Off (execution, ROM), On (RAM)	No impact	Off
Bitfield declarator type specifier	No impact	No impact	Target dependent	No impact	`uint_T`

Mapping of Application Requirements to the Optimization Pane: Signals and Parameters tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Simplify array indexing	No impact	No impact	No impact	No impact	Off
Pass reusable subsystem outputs as	No impact	No impact	No impact (execution), `Structure reference` (ROM), `Individual arguments` (RAM)	No impact	`Structure reference`

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
Eliminated / virtual blocks	On	On	No impact	On	Off

Mapping of Application Requirements to the Code Generation Pane: Report Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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

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 (MPT objects only)	On	On	No impact	No impact	Off
Custom comments function	Any valid file name	Any valid file name	No impact	No impact	''

Mapping of Application Requirements to the Code Generation Pane: Comments Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Stateflow object descriptions	On	On	No impact	No impact	Off
Requirements in block comments	On	On	No impact	On	Off

Mapping of Application Requirements to the Code Generation Pane: Symbols Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Global variables	No impact	Any valid combination of tokens	No impact	`$R$N$M`	`$R$N$M`
Global types	No impact	Any valid combination of tokens	No impact	`$N$R$M`	`&N$R$M`
Field name of global types	No impact	Any valid combination of tokens	No impact	`$N$M`	`$N$M`
Subsystem methods	No impact	Any valid combination of tokens	No impact	`$R$N$M$F`	`$R$N$M$F`
Subsystem method arguments	No impact	Any valid combination of tokens	No impact	`rtu_$N$M` or `rty_$N$M`	`rtu_$N$M` or `rty_$N$M`
Local temporary variables	No impact	Any valid combination of tokens	No impact	`$N$M`	`$N$M`

Mapping of Application Requirements to the Code Generation Pane: Symbols Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Local block output variables	No impact	Any valid combination of tokens	No impact	`rtb_$N$M`	`rtb_$N$M`
Constant macros	No impact	Any valid combination of tokens	No impact	`$R$N$M`	`$R$N$M`
Minimum mangle length	No impact	1	No impact	No impact	1
Generate scalar inlined parameters as	No impact	`Macros`	`Literals`	No impact	`Literals`
#define naming	No impact	`Force uppercase`	No impact	No impact	`None`
Parameter naming	No impact	`Force uppercase`	No impact	No impact	`None`
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-line) for integer only	No impact	On (GUI), `'off'` (command-line)
Support complex numbers	No impact	No impact	Off for real only	No impact	On
Support 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	Specifying `User defined` and a low value for **Maximum word length** reduces the size of the generated	Use default	`System defined`

Mapping of Application Requirements to the Code Generation Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
			file `rtwtypes.h`		
Maximum word length	No impact	No impact	Smaller values reduce the size of the generated file `rtwtypes.h`	Use default	256
GRT compatible call interface	No impact	Off	Off (execution, ROM), No impact (RAM)	Off	Off
Single output/update function	On	On	On	On	On
Terminate function required	No impact	No impact	Off (execution, ROM), No impact (RAM)	Off	On
Generate reusable code	No impact	No impact	Set for single instance	No impact	Off
Reusable code error diagnostic	`Warning` or `Error`	No impact	`None`	No impact	`Error`
Pass root-level I/O as	No impact	No impact	No impact	No impact	`Individual arguments`

Mapping of Application Requirements to the Code Generation Pane: Interface Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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 operator new for referenced model object registration	No impact	No impact	On	Off	Off
Generate preprocessor conditionals	No impact	No impact	No impact	No impact	`Use local settings`
Suppress error status in real-time model data structure	Off	No impact	On	On	Off
"Combine signal/state structures"	Off	No impact	No impact	On	No impact

Mapping of Application Requirements to the Code Generation Pane: SIL and PIL Verification Tab

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
Create block	On	No impact	No impact	No impact	Off
Enable portable word sizes	On	No impact	Off	Off	Off
Measure task execution time	On	On	Off	No impact	Off
Measure function execution times	On	On	Off	No impact	Off
Workspace variable	No impact	Any valid string	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 for precedence)`	`Maximum (Specify precedence with parentheses)`	`Nominal (Optimize for readability)`
Preserve operand order in expression	On	On	Off	On	Off

Mapping of Application Requirements to the Code Generation Pane: Code Style Tab (Continued)

Configuration Parameter	Debugging	Traceability	Efficiency	Safety Precaution	Factory Default
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

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	Any valid value	No impact	No impact	Auto
Data definition filename	No impact	Any valid value	No impact	No impact	`global.c`
Data declaration	No impact	Any valid value	No impact	No impact	Auto
Data declaration filename	No impact	Any valid value	No impact	No impact	`global.h`
#include file delimiter	No impact	Any valid value	No impact	No impact	`off`
#include file delimiter	No impact	Any valid value	No impact	No impact	Auto
Signal display level	No impact	Any valid integer	No impact	No impact	10
Parameter tune level	No impact	Any valid integer	No impact	No impact	10
File packaging format	No impact	No impact	No impact	No impact	`Modular`

Mapping of Application Requirements to the 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	Any 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`
Execution	No impact	No impact	No impact	No impact	`Default`
Constants	No impact	No impact	No impact	No impact	`Default`
Inputs/Outputs	No impact	No impact	No impact	No impact	`Default`
Internal data	No impact	No impact	No impact	No impact	`Default`
Parameters	No impact	No impact	No impact	No impact	`Default`
Validation results	No impact	No impact	No impact	No impact	Package and memory sections found.

Parameter Command-Line Information Summary

The following tables list Embedded Coder parameters that you can use to tune model and target configurations. The table provides brief descriptions, valid values (bold type highlights defaults), and a mapping to Configuration

Parameter dialog box equivalents. For descriptions of the panes and options in that dialog box, see Configuration Parameters.

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.

For information about Simulink parameters, see "Configuration Parameters Dialog Box" in the Simulink documentation. For information about Simulink Coder parameters, see "Configuration Parameters for Simulink Models" in the Simulink Coder documentation. For information on using `get_param` and `set_param` to tune the parameters for various model configurations, see "Tuning Parameters". See "Using Configuration Wizard Blocks" for information on using Configuration Wizard features.

Note Parameters that are specific to the ERT target or targets based on the ERT target, Stateflow, or the Simulink® Fixed Point™ product are marked with (ERT), (Stateflow), and (Simulink Fixed Point), respectively. To set the values of parameters marked with (ERT), you must specify an ERT or ERT-based target for your configuration set. Also, note that the default setting for a parameter might vary for different targets. Parameters marked with (ERT) are listed with ERT target defaults.

Command-Line Information: Optimization Pane: General tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`NoFixptDivByZeroProtection` (ERT) (Simulink Fixed Point) **`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**	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. 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. • 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

Command-Line Information: Optimization Pane: General tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		a Model block that references a model that has the **Optimize initialization code for model reference** parameter set to `on`. • If you set the **Optimize initialization code for model reference** parameter to `off` in a model that includes a Model block that directly references a submodel, do not set the **Optimize initialization code for model reference** parameter for the submodel to `on`.
`UseSpecifiedMinMax` (ERT) *`string`* - **`off`**, `on`	**Optimize using the specified minimum and maximum values**	Use the specified minimum and maximum values, such as block Output minimum and
`ZeroExternalMemoryAtStartup` (ERT) `off`, **`on`**	**Remove root level I/O zero initialization**	Suppress code that initializes root-level I/O data structures to zero.
`ZeroInternalMemoryAtStartup` (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.
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.
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) *string* - `off`, **`on`**	**Code Generation > General > Ignore custom storage classes**	Treat custom storage classes as `'Auto'`.
`IgnoreTestpoints` (ERT) *string* - **`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) *string* - **`off`**, `on`	**Code Generation > Report > Model-to-code**	Includes model-to-code traceability support in the generated HTML report.
`IncludeHyperlinkInReport` (ERT) *string* - **`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.

Command-Line Information: Code Generation Pane: Report Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateTraceReport` (ERT) *string* - **`off`**, `on`	**Code Generation > Report > Eliminated / virtual blocks**	Include summary of eliminated and virtual blocks in Code Generation report.
`GenerateTraceReportSl` (ERT) *string* - **`off`**, `on`	**Code Generation > Report > Traceable Simulink blocks**	Include summary of Simulink blocks in Code Generation report.
`GenerateTraceReportSf` (ERT) *string* - **`off`**, `on`	**Code Generation > Report > Traceable Stateflow objects**	Include summary of Stateflow objects in Code Generation report.
`GenerateTraceReportEml` (ERT) *string* - **`off`**, `on`	**Code Generation > Report > Traceable MATLAB functions**	Include summary of MATLAB functions in Code Generation report.

Command-Line Information: Code Generation Pane: Comments Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomCommentsFcn` (ERT) *string* - `''`	**Code Generation > Comments > Custom comments function**	Specify the filename of the MATLAB or TLC function that adds the custom comment.
`EnableCustomComments` (ERT) *string* - **`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.

Command-Line Information: Code Generation Pane: Comments Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`InsertBlockDesc` (ERT) *string* - **`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) *string* - **`off`**, `on`	**Code Generation > Comments > Requirements in block comments**	Include specified requirements in the generated code as a comment.
`SFDataObjDesc` (ERT) *string* - **`off`**, `on`	**Code Generation > Comments > Stateflow object descriptions**	Insert Stateflow object descriptions into the generated code as a comment.
`SimulinkDataObjDesc` (ERT) *string* - **`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) *string* - **`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

Command-Line Information: Code Generation Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		`$N` - Name of object `$A` - Data type acronym
`CustomSymbolStrFcn` (ERT) *string* - **`$R$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 `$F` - Subsystem method name
`CustomSymbolStrFcnArg`(ERT) *string* - **`rtu_$N$M`** or **`rty_$N$M`**	**Code Generation > Symbols > Subsystem method arguments**	Specify a symbol format rule for subsystem method arguments. The rule can contain valid C identifier characters and the following macros: `$I` — `u` if the argument is an input or `y` if the argument is an output `$M` - Mangle `$N` - Name of object

Command-Line Information: Code Generation Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrField` (ERT) *string* - **NM**	**Code Generation > Symbols > Field name of global types**	Specify a symbol format rule for field name of global types. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$N` - Name of object `$H` - System hierarchy number `$A` - Data type acronym
`CustomSymbolStrGlobalVar` (ERT) *string* - **RN$M**	**Code Generation > Symbols > Global variables**	Specify a symbol format rule for global variables. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$R` - Root model name `$N` - Name of object
`CustomSymbolStrMacro` (ERT) *string* - **RN$M**	**Code Generation > Symbols > Constant macros**	Specify a symbol format rule for constant macros. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$R` - Root model name `$N` - Name of object

Command-Line Information: Code Generation Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSymbolStrTmpVar` (ERT) *string* - **NM**	**Code Generation > Symbols > Local temporary variables**	Specify a symbol format rule for local temporary variables. The rule can contain valid C identifier characters and the following macros: `$M` - Mangle `$R` - Root model name `$N` - Name of object
`CustomSymbolStrType` (ERT) *string* - **NR$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
`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 all `#define` names.

Command-Line Information: Code Generation Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`IncDataTypeInIds` (ERT) **`off`**, `on`	**Code Generation > Symbol > Include data type acronym in identifiers**	Include acronyms that express data types in signal and work vector identifiers. For example, `'rtB.i32_signame'` identifies a 32-bit integer block output signal named `'signame'`.
`IncHierarchyInIds` (ERT) **`off`**, `on`	**Code Generation > Symbols > Include system hierarchy number in identifiers**	Include the system hierarchy number in variable identifiers. For example, `'s3_'` is the system hierarchy number in `rtB.s3_signame` for a block output signal named `'signame'`. Including the system hierarchy number in identifiers improves the traceability of generated code. To locate the subsystem in which the identifier resides, type `hilite_system('<S3>')` at the MATLAB prompt. The argument specified with `hilite_system` requires an uppercase `S`.
`InlinedPrmAccess` (ERT) string - **`Literals`**, `Macros`	**Code Generation > Symbols > Generate scalar inlined parameters as**	Specify whether inlined parameters are coded as numeric constants or macros. Specify `Macros` for more efficient code.

Command-Line Information: Code Generation Pane: Symbols Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MangleLength` (ERT) `int` - **`1`**	**Code Generation > Symbols > Minimum mangle length**	Specify the minimum number of characters to be used for name mangling strings generated and applied to symbols to avoid name collisions. A larger value reduces the chance of identifier disturbance when you modify the model.
`ParamNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Code Generation > Symbols > Parameter naming**	Select a rule that changes spelling of all parameter names.
`PrefixModelToSubsysFcnNames` (ERT) `off`, **`on`**	**Code Generation > Symbols > Prefix model name to global identifiers**	Add the model name as a prefix to subsystem function names for all code formats. When appropriate for the code format, also add the model name as a prefix to top-level functions and data structures. This prevents compiler errors due to name clashes when combining multiple models.
`SignalNamingRule` (ERT) string - **`None`**, `UpperCase`, `LowerCase`, `Custom`	**Code Generation > Symbols > Signal naming**	Specify a rule the code generator is to use that changes spelling of all signal names.

Command-Line Information: Code Generation Pane: Interface Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CombineOutputUpdateFcns` (ERT) string - `off`, **`on`**	**Code Generation > Interface > Single output/update function**	Generate a model's output and update routines into a single-step function.
`ERTMaxMultiwordLength` (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 `rtwtypes.h`. Specifying 0 provides you complete control over type definitions for multiword data types in generated code.
`ERTMultiwordTypeDef` (ERT) string - **`System defined`**, `User defined`	**Code Generation > Interface > Multiword type definitions**	Specify whether to use system-defined or user-defined type definitions for multiword data types in generated code.
`GenerateDestructor` (ERT) string - `off`, **`on`**	**Code Generation > Interface > Generate destructor**	Generate a destructor for the model class in `C++ (Encapsulated)` model code.
`GenerateExternalIOAccess-Methods` (ERT) string - **`None`**, `Method`, `Inlined method`, `Structure-based method`, `Inlined structure-based method`	**Code Generation > Interface > External I/O access**	Specify whether to generate access methods for root-level I/O signals for the `C++ (Encapsulated)` model class.

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
GenerateInternalMember-AccessMethods (ERT) string - **None**, Method, Inlined method	**Code Generation > Interface > Internal data access**	Specify whether to generate access methods for internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states for the C++ (Encapsulated) model class.
GenerateParameterAccess-Methods (ERT) string - **None**, Method, Inlined method	**Code Generation > Interface > Block parameter access**	Specify whether to generate access methods for block parameters for the C++ (Encapsulated) model class.
GeneratePreprocessor-Conditionals (ERT) string - **Use local settings**, Enable all, Disable all	**Code Generation > Interface > Generate preprocessor conditionals**	Specify whether to generate preprocessor conditionals locally for each Model block containing variants or globally for all Model blocks in a model.
GRTInterface (ERT) string - **off**, on	**Code Generation > Interface > GRT compatible call interface**	Include a code interface (wrapper) that is compatible with the GRT target.
IncludeMdlTerminateFcn (ERT) string - off, **on**	**Code Generation > Interface > Terminate function required**	Generate a terminate function for the model.

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`InternalMemberVisibility` (ERT) string - `public`, **`private`**, `protected`	**Code Generation > Interface > Internal data visibility**	Specify whether to generate internal data structures such as Block I/O, DWork vectors, Run-time model, Zero-crossings, and continuous states as `public`, `private`, or `protected` data members of the `C++ (Encapsulated)` model class.
`MultiInstanceErrorCode` (ERT) string - `None`, `Warning`, **`Error`**	**Code Generation > Interface > Reusable code error diagnostic**	Specify the error diagnostic behavior for cases when data defined in the model violates the requirements for generation of reusable code.
`MultiInstanceERTCode` (ERT) string - **`off`**, `on`	**Code Generation > Interface > Generate reusable code**	Specify whether to generate reusable, reentrant code.
`ParameterMemberVisibility` (ERT) string - `public`, **`private`**, `protected`	**Code Generation > Interface > Block parameter visibility**	Specify whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the `C++ (Encapsulated)` model class.

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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`	**Code Generation > Interface > Pass root-level I/O as**	Specify how the code generator is to pass root-level I/O data into a reusable function.
`SupportAbsoluteTime` (ERT) string - **`off`**, `on`	**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) string - **`off`**, `on`	**Code Generation >Interface > variable-size signals**	Specify whether to generate code for models that use variable-size signals.

Command-Line Information: Code Generation Pane: Interface Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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**	Specify whether to 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.
`UseOperatorNewForModelRef-Registration` (ERT) string - **`off`**, `on`	**Code Generation > Interface > Use operator new for referenced model object registration**	For a model containing Model blocks, specify whether generated code should use the operator `new`, during model object registration, to instantiate objects for referenced models configured with a C++ encapsulation interface.

Command-Line Information: Code Generation Pane: SIL and PIL Verification Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
CodeExecutionProfileVariable (ERT) *string* - **executionProfile**	**Code Generation > SIL and PIL Verification > Workspace variable**	Specify workspace variable that collects measurements and allows viewing and analysis of execution profiles.
CodeExecutionProfiling (ERT) string - **off**, on	**Code Generation > SIL and PIL Verification > Measure task execution time**	Specify whether to collect execution time profiles for tasks in generated code.
CodeProfilingInstrumentation (ERT) string - **off**, on	**Code Generation > SIL and PIL Verification > Measure function execution times**	Specify whether to collect execution time profiles for functions in code generated from the model.
CodeProfilingSaveOptions (ERT) string - **Summary data only**, All measurement and analysis data	**Code Generation > SIL and PIL Verification > Save options**	Specify whether to save all code profiling measurement and analysis data to base workspace.
CreateSILPILBlock (ERT) string - **None**, SIL, PIL	**Code Generation > SIL and PIL Verification > Create block**	Create SIL or PIL block to allow verification of source or object code generated from subsystem or top-model components.

Command-Line Information: Code Generation Pane: SIL and PIL Verification Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`GenerateErtSFunction` (ERT) string - **`off`**, `on` *Will be removed in a future release. Replaced by* `CreateSILPILBlock`	**Code Generation > SIL and PIL Verification > Create block**	Wrap the generated code inside an S-Function block. This allows you to validate the generated code in a Simulink model.
`PortableWordSizes` (ERT) string - **`off`**, `on`	**Code Generation > SIL and PIL Verification > Enable portable word sizes**	Specify that model code should be generated with conditional processing macros that allow the same generated source code files to be used both for software-in-the-loop (SIL) testing on the host platform and for production deployment on the target platform.

Command-Line Information: Code Generation Pane: Code Style Tab

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.

Command-Line Information: Code Generation Pane: Code Style Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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.
`SuppressUnreachableDefault-Cases` (ERT) string - **`off`**, `on`	**Code Generation > Code Style > Suppress generation of default cases for Stateflow switch statements if unreachable**	Control whether to always generate default cases for switch-case statements in the code for Stateflow charts.

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.

Command-Line Information: Code Generation Pane: Templates Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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 to the file `ert_main.c` or `ert_main.cpp`.
`TargetOS` (ERT) string - **`BareBoardExample`**, `VxWorksExample`	**Code Generation > Templates > Target operating system**	Specify the target operating system for the example main `ert_main.c` or `ert_main.cpp`. `BareBoardExample` is a generic example that assumes no operating system. `VxWorksExample` is tailored to the VxWorks[9] real-time operating system.

9. 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 `.c` or `.cpp` 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 `.c` or `.cpp` file that contains global data references.
`GlobalDataDefinition` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateSourceFile`	**Code Generation > Code Placement > Data definition**	Select the `.c` or `.cpp` file where variables of global scope are defined.
`GlobalDataReference` (ERT) string - **`Auto`**, `InSourceFile`, `InSeparateHeaderFile`	**Code Generation > Data Placement > Data declaration**	Select the `.h` file where variables of global scope are declared (for example, `extern real_T globalvar;`).
`IncludeFileDelimiter` (ERT) string - **`Auto`**, `UseQuote`, `UseBracket`	**Code Generation > Code Placement > #include file delimiter**	Specify the delimiter to be used for all data objects that do not have a delimiter specified in the `IncludeFile` property.
`EnableDataOwnership` string - **`off`**, `on`	**Code Generation > Code Placement > Use owner from mpt object for data definition placement**	Specify whether the model uses the ownership setting of an mpt data object for data definition in code generation.
`ModuleNamingRule` (ERT) string - **`Unspecified`**, `SameAsModel`	**Code Generation > Code Placement > Use owner from mpt object for data definition placement**	Specify whether the model uses the ownership setting of an mpt data object for data definition in code generation.

Command-Line Information: Code Generation Pane: Code Placement Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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: • Initialize/Start functions • Terminate functions
`MemSecFuncExecute` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Code Generation > Memory Sections > Execution**	Apply memory sections to: • Step functions • Run-time initialization functions • Derivative functions • Enable functions • Disable functions
`MemSecDataConstants` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Code Generation > Memory Sections > Constants**	Apply memory sections to: • Constant parameters • Constant block I/O • Zero representation
`MemSecDataIO` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Code Generation > Memory Sections > Inputs/Outputs**	Apply memory sections to: • Root inputs • Root outputs

Command-Line Information: Code Generation Pane: Memory Sections Tab (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`MemSecDataInternal` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Code Generation > Memory Sections > Internal data**	Apply memory sections to: • Block I/O • DWork vectors • Run-time model • Zero-crossings
`MemSecDataParameters` (ERT) string - **`Default`**, `MemConst`, `MemVolatile`, `MemConstVolatile`	**Code Generation > Memory Sections > Parameters**	Apply memory sections to: • Parameters

Command-Line Information: Not in GUI

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CPPClassGenCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to generate and configure C++ encapsulation interfaces to model code. Default is `off` for custom and non-ERT targets and `on` for ERT (`ert.tlc`) targets. (For more information, see "Supporting C++ Encapsulation Interface Control".)
`ERTFirstTimeCompliant` (ERT) string - `off`, **`on`**	Not available	Set in `SelectCallback` for a target to indicate whether the target supports the ability to control inclusion of the `firstTime` argument in the

Command-Line Information: Not in GUI (Continued)

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
		*model*_initialize function generated for a Simulink model. Default is off for custom and non-ERT targets and on for ERT targets. (For more information, see "Supporting firstTime Argument Control".)
IncludeERTFirstTime (ERT) string - **off**, on **Note** The value of IncludeERTFirstTime is meaningful only if the target configuration parameter ERTFirstTimeCompliant is set to on for your selected target.	Not available	Specify whether code generation software is to include the firstTime argument in the *model*_initialize function generated for a Simulink model.
ModelStepFunctionPrototype-ControlCompliant (ERT) string - off, **on**	Not available	Set in SelectCallback for a target to indicate whether the target supports the ability to control the function prototypes of initialize and step functions that are generated for a Simulink model. Default is off for non-ERT targets and on for ERT targets. (For more information, see "Supporting C Function Prototype Control".)

Index

C

D

E

F

G

H

I

L

M

V

W