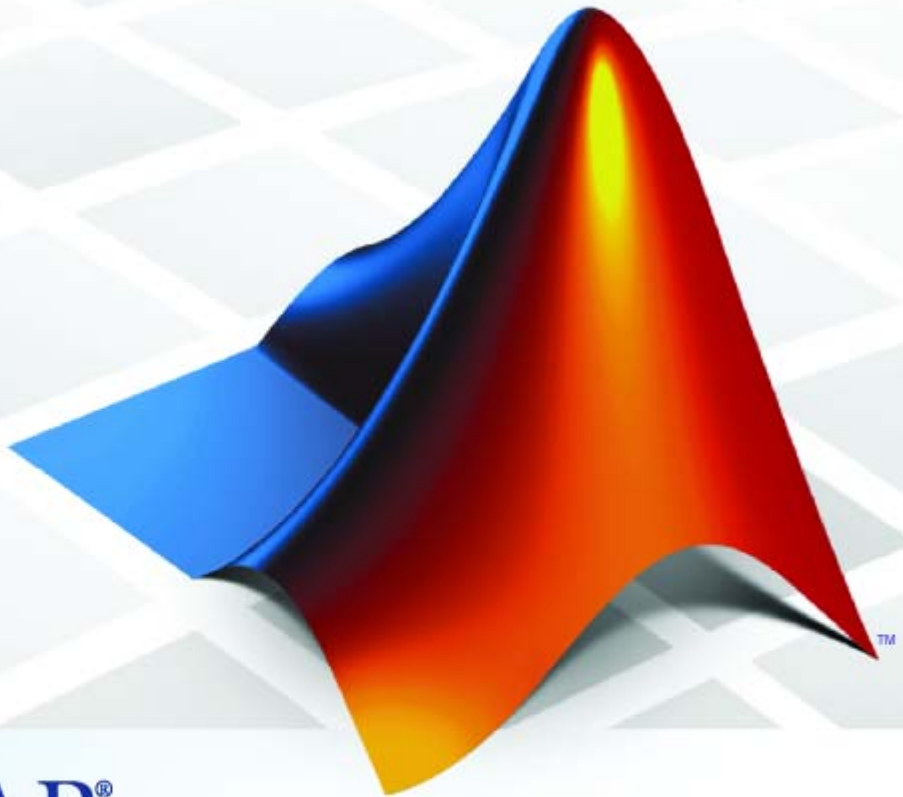


# Embedded IDE Link™ 4

## User's Guide



MATLAB®

## How to Contact The MathWorks



[www.mathworks.com](http://www.mathworks.com)  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab)  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html)

Web  
Newsgroup  
Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com)  
[bugs@mathworks.com](mailto:bugs@mathworks.com)  
[doc@mathworks.com](mailto:doc@mathworks.com)  
[service@mathworks.com](mailto:service@mathworks.com)  
[info@mathworks.com](mailto:info@mathworks.com)

Product enhancement suggestions  
Bug reports  
Documentation error reports  
Order status, license renewals, passcodes  
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 IDE Link™ User's Guide*

© COPYRIGHT 2010 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### Revision History

March 2010      Online only      New for Version 4.1 (Release 2010a)

## Getting Started

### 1

<b>Product Overview</b> .....	1-2
Overview .....	1-2
Key Features .....	1-2
Introduction .....	1-3
Generating IDE Projects .....	1-4
Generating Makefiles .....	1-4
Automating IDE Tasks .....	1-4
Verifying Models Running on Targets .....	1-5
Optimizing Models .....	1-5
<b>Using this Guide</b> .....	1-7
<b>Installation and Configuration</b> .....	1-9

## Preparing Models for Embedded Deployment

### 2

<b>Setting Target Preferences</b> .....	2-2
What are Target Preferences Blocks? .....	2-2
Locating a Target Preferences Block .....	2-3
Configuring a Target Preferences Block for a Supported Processor .....	2-3
Adding a Target Preferences Block to Your Model .....	2-4
Examples of Configuring Target Preferences .....	2-5
<b>Setting Configuration Parameters for Embedded IDE   Link</b> .....	2-6
What are Configuration Parameters? .....	2-6
Setting Model Configuration Parameters .....	2-6

<b>Working with Block Libraries</b> .....	<b>2-15</b>
<b>Simulink Models and Targeting</b> .....	<b>2-16</b>
Creating Your Simulink Model for Targeting .....	<b>2-16</b>
Blocks to Avoid in Your Models .....	<b>2-17</b>

## Generating IDE Projects

# 3

<b>Introducing Project Generator</b> .....	<b>3-2</b>
<b>Project Generation</b> .....	<b>3-3</b>
<b>Schedulers and Timing</b> .....	<b>3-4</b>
Configuring Models for Asynchronous Scheduling .....	<b>3-4</b>
Cases for Using Asynchronous Scheduling .....	<b>3-5</b>
Using Scheduling Blocks to Control Code Execution .....	<b>3-7</b>
Comparing Synchronous and Asynchronous Interrupt Processing .....	<b>3-7</b>
Using Synchronous Scheduling .....	<b>3-9</b>
Using Asynchronous Scheduling .....	<b>3-10</b>
Multitasking Scheduler Examples .....	<b>3-10</b>
<b>Project Generator Tutorial</b> .....	<b>3-23</b>
Creating the Model .....	<b>3-24</b>
Adding the Target Preferences Block to Your Model .....	<b>3-24</b>
Specify Configuration Parameters for Your Model .....	<b>3-26</b>
<b>Setting Code Generation Parameters for Processors</b> ..	<b>3-29</b>
<b>Using Custom Source Files in Generated Projects</b> .....	<b>3-32</b>
Preparing to Replace Generated Files With Custom Files .....	<b>3-32</b>
Replacing Generated Source Files with Custom Files When You Generate Code .....	<b>3-34</b>

<b>Optimizing Embedded Code with Target Function</b>	
<b>Libraries</b> .....	<b>3-36</b>
About Target Function Libraries and Optimization .....	<b>3-36</b>
Using a Processor-Specific Target Function Library to Optimize Code .....	<b>3-38</b>
Process of Determining Optimization Effects Using Real-Time Profiling Capability .....	<b>3-39</b>
Reviewing Processor-Specific Target Function Library Changes in Generated Code .....	<b>3-40</b>
Reviewing Target Function Library Operators and Functions .....	<b>3-42</b>
Creating Your Own Target Function Library .....	<b>3-42</b>
<b>Model Reference</b> .....	<b>3-43</b>
How Model Reference Works .....	<b>3-43</b>
Using Model Reference .....	<b>3-44</b>
Configuring processors to Use Model Reference .....	<b>3-46</b>

## Generating Makefiles

# 4

<b>Using Makefiles to Generate and Build Software</b> .....	<b>4-2</b>
Overview .....	<b>4-2</b>
Configuring Your Model to Use Makefiles .....	<b>4-2</b>
Choosing an XMakefile Configuration .....	<b>4-3</b>
Building Your Model .....	<b>4-5</b>
<b>Making an XMakefile Configuration Operational</b> .....	<b>4-6</b>
<b>Example: Creating an XMakefile Configuration for the Intel Compiler</b> .....	<b>4-7</b>
Overview .....	<b>4-7</b>
Create a Configuration .....	<b>4-7</b>
Modify the Configuration .....	<b>4-9</b>
Test the Configuration .....	<b>4-12</b>
<b>XMakefile User Configuration Dialog Box</b> .....	<b>4-17</b>
Active .....	<b>4-17</b>
Make Utility .....	<b>4-19</b>

Compiler .....	4-20
Linker .....	4-21
Archiver .....	4-21
Pre-build .....	4-22
Post-build .....	4-23
Execute .....	4-23
Tool Directories .....	4-24

## Verifying Generated Code

### 5

<b>What Is Verification?</b> .....	5-2
<b>Verifying Generated Code via Processor-in-the-Loop</b> ..	5-3
What is Processor-in-the-Loop? .....	5-3
Using the Top-Model PIL Approach .....	5-5
Using the PIL Block Approach .....	5-6
Definitions .....	5-8
Other Aspects of PIL .....	5-9
PIL Issues and Limitations .....	5-9
<b>Profiling Code Execution in Real-Time</b> .....	5-10
Overview .....	5-10
Profiling Execution by Tasks .....	5-11
Profiling Execution by Subsystems .....	5-13
<b>System Stack Profiling</b> .....	5-18
Overview .....	5-18
Profiling System Stack Use .....	5-20

## Block Reference

### 6

<b>Block Library: idelinklib_common</b> .....	6-2
---	-----

7

Function Reference

8

Setup .....	8-2
Constructor .....	8-3
File and Project Operations .....	8-4
Processor Operations .....	8-5
Debug Operations .....	8-6
Data Manipulation .....	8-7
Status Operations .....	8-8
<b>Grouped by IDE</b> .....	8-9
Altium TASKING .....	8-9
Analog Devices™ VisualDSP++ .....	8-9
Eclipse IDE .....	8-11
Green Hills® MULTI .....	8-12
Texas Instruments Code Composer Studio .....	8-14

## Configuration Parameters

---

<b>Embedded IDE Link Pane</b> .....	<b>10-2</b>
Overview .....	10-4
Build format .....	10-5
Build action .....	10-7
Overrun notification .....	10-10
Function name .....	10-12
PIL block action .....	10-13
Configuration .....	10-15
Compiler options string .....	10-17
Linker options string .....	10-19
System stack size (MAUs) .....	10-21
System heap size (MAUs) .....	10-23
Profile real-time execution .....	10-24
Profile by .....	10-26
Number of profiling samples to collect .....	10-28
Maximum time allowed to build project (s) .....	10-30
Maximum time allowed to complete IDE operations (s) ...	10-32
Export IDE link handle to base workspace .....	10-33
IDE link handle name .....	10-35
Source file replacement .....	10-36

---

## Index



# Getting Started

---

- “Product Overview” on page 1-2
- “Using this Guide” on page 1-7
- “Installation and Configuration” on page 1-9

## Product Overview

In this section...
“Overview” on page 1-2
“Key Features” on page 1-2
“Introduction” on page 1-3
“Generating IDE Projects” on page 1-4
“Generating Makefiles” on page 1-4
“Automating IDE Tasks” on page 1-4
“Verifying Models Running on Targets” on page 1-5
“Optimizing Models” on page 1-5

### Overview

Embedded IDE Link™ connects MATLAB® and Simulink® with embedded software development environments. Embedded IDE Link lets you generate, build, test, and optimize embedded code for prototyping or production. It automates debugging, project generation, and verification of object code executing on an embedded processor or on the instruction set simulator provided by your IDE. You can reuse MATLAB code or Simulink models as a test bench for Processor-in-the-Loop testing of manually written or generated code.

### Key Features

- Automates software debugging, verification, and analysis using MATLAB and Simulink to assess code generated automatically or manually
- Enables processor-in-the-loop (PIL) testing of target-compiled object code by reusing your Simulink model as an interactive embedded software test bench
- Provides profiling capabilities for real-time execution and stack usage plus custom cache configuration and memory mapping for some IDEs

- Provides target-specific code optimization libraries with customization capabilities
- Generate a complete standalone project or a target compiled library from your model for execution on embedded processors
- Facilitates transition from your modeling environment to code coverage structural analysis, MISRA C code checking, and additional analyses provided by your IDE
- Supports IDEs and processors from third-party vendors such as Altium™, Analog Devices™, ARM®, Freescale™, Green Hills Software™, Infineon®, Renesas®, STMicroelectronics®, and Texas Instruments®

## Introduction

You can use Embedded IDE Link™ to build software for embedded systems from Simulink® models via the following IDEs:

- Altium® TASKING®
- Analog Devices™ VisualDSP++®
- Eclipse™
- Green Hills® MULTI®
- Texas Instruments' Code Composer Studio™

Embedded IDE Link:

- Can generate makefiles, which you can use to build software automatically without using an IDE.
- Can generate software for operating systems such as Linux and Windows.
- Includes a Project Generator component that can build software automatically via IDEs and load it onto the target.
- Includes an Automation Interface component, which is a MATLAB API you can use to automate complex tasks via MATLAB scripts.
- Provides tools for verification and optimization activities.

## **Generating IDE Projects**

- Automated project-based build process  
Automatically create and build projects for code generated by the Real-Time Workshop® or Real-Time Workshop® Embedded Coder™ products.
- Highly customizable code generation  
Use Embedded IDE Link software with any Real-Time Workshop System Target File (STF) to generate target-specific and optimized code.
- Highly customizable build process  
Support for multiple IDEs provides a route to many different target hardware platforms.
- Automated download and debugging  
Debug generated code in the IDE, using either the instruction set simulator or real hardware.

## **Generating Makefiles**

- Automated makefile-based build process  
Automatically create makefiles and build code generated by the Real-Time Workshop or Real-Time Workshop Embedded Coder products.
- Highly customizable code generation  
Use Embedded IDE Link software with any Real-Time Workshop System Target File (STF) to generate target-specific and optimized code.
- Highly customizable build process  
Support for multiple software build toolchains provides a route to many different target hardware platforms.
- New software build toolchains  
Create profiles to generate makefiles for new software toolchains.

## **Automating IDE Tasks**

- Provides a MATLAB® API for automating complex tasks in the IDE.

Automate complex tasks in the IDE by writing MATLAB scripts to communicate with the IDE.

For example, you could

- Automate project creation, including adding source files, include paths, and preprocessor defines.
- Configure batch building of projects.
- Launch a debugging session.

## Verifying Models Running on Targets

- Processor-in-the-loop (PIL) cosimulation  
Verify generated code running in an instruction set simulator or real target environment.
- C Code Coverage  
Use C code instruction coverage metrics during PIL cosimulation to refine test cases.
- Execution Profiling  
Use execution profiling metrics during PIL cosimulation to establish the timing requirements of your algorithm.
- Stack Profiling  
Use stack profiling metrics for PIL cosimulation or real-time applications to verify the amount of memory allocated for stack usage is sufficient.
- Bidirectional Traceability Between Model and Code  
Navigate to and from the generated code for a given Simulink block.

## Optimizing Models

- Compiler / Linker Optimization Settings  
Use Template Projects to control compiler and linker optimization settings.
- Target Memory Placement / Mapping  
Use Template Projects to configure the target memory map.

- Execution Profiling

Use execution profiling during PIL cosimulation to guide optimization of your algorithms.

- Stack Profiling

Use stack profiling metrics for PIL cosimulation or real-time applications to optimize the amount of stack memory required for an application.

## Using this Guide

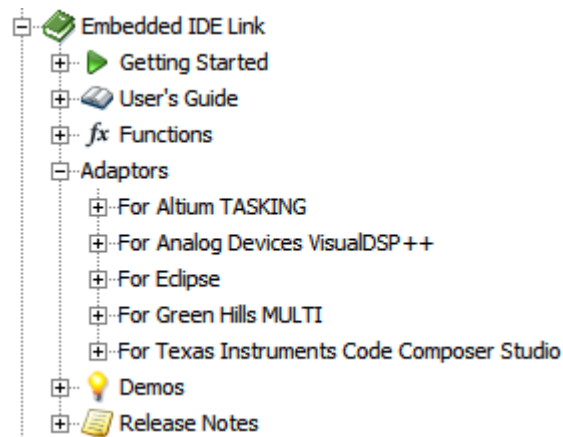
The Embedded IDE Link product works with a number of IDEs. The structure of this documentation provides both general product information, and information that applies to specific IDEs.

---

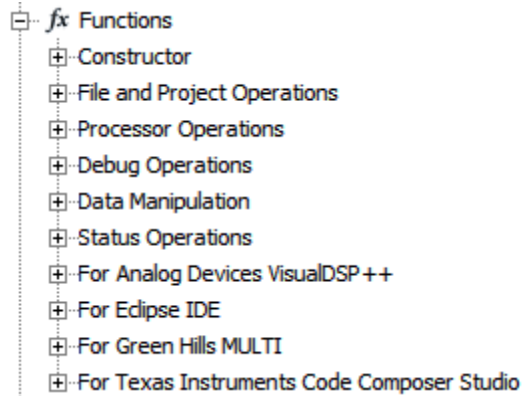
**Note** It is important for you to understand the documentation structure so you can find both the general and the specific information you need.

---

In the Embedded IDE Link book, the **Getting Started** and **User Guide** sections provide general information. The IDE-specific information is located in the **Supported IDEs** sections.



The **Functions** section presents all of the functions, categorized by type and by IDE.



Four of the five sections under **Supported IDEs** contain documentation for previous stand-alone products. They are:

- **For Altium TASKING** used to be the documentation for Embedded IDE Link TS.
- **For Analog Devices VisualDSP++** used to be the documentation for Embedded IDE Link VS
- **For Green Hills MULTI** used to be the documentation for Embedded IDE Link MU
- **For Texas Instruments Code Composer Studio** used to be the documentation for Embedded IDE Link CC

We are in the process of refactoring these four sections.

---

**Note** If you are using the TASKING IDE, only refer to the content in **For Altium TASKING**. We have not incorporated this content into the Embedded IDE Link Getting Started and User's Guide sections.

---



## Installation and Configuration

For installation and configuration instructions, including software requirements, see the following topic for your IDE:

- Altium TASKING: “Getting Started”
- Analog Devices VisualDSP++: “Getting Started”
- Eclipse IDE: “Getting Started”
- Green Hills MULTI: “Getting Started”
- Texas Instruments<sup>™</sup>Code Composer Studio<sup>™</sup>: “Getting Started”



# Preparing Models for Embedded Deployment

---

- “Setting Target Preferences” on page 2-2
- “Setting Configuration Parameters for Embedded IDE Link” on page 2-6
- “Working with Block Libraries” on page 2-15
- “Simulink Models and Targeting” on page 2-16

# Setting Target Preferences

In this section...
“What are Target Preferences Blocks?” on page 2-2
“Locating a Target Preferences Block” on page 2-3
“Configuring a Target Preferences Block for a Supported Processor” on page 2-3
“Adding a Target Preferences Block to Your Model” on page 2-4
“Examples of Configuring Target Preferences” on page 2-5

The following IDE's and processor families use target preferences blocks. The information in this section applies to them:

- Texas Instruments Code Composer Studio, C2000™, C5000™, and C6000™
- Analog Devices VisualDSP++®, and Blackfin®
- Eclipse IDE
- Green Hills MULTI®

The following IDE's and processor families do not use target preferences blocks:

- Freescale MPC5xx
- Altium TASKING
- Infineon C166®

The information in this section does not apply to them.

## What are Target Preferences Blocks?

A target preferences block describes the environment for which you are generating code. The block includes information about the processor, hardware settings, operating system, memory mapping, and code generation features. The Real-Time Workshop, Embedded IDE Link, and Simulink products use this information to generate code from your model.

## Locating a Target Preferences Block

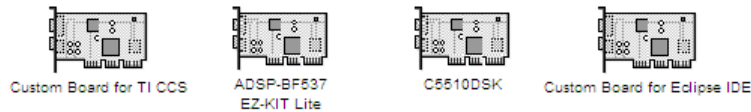
Target preferences blocks are located in:

- The Target Support Package™ block libraries for Supported Processors.
- The Embedded IDE Link block libraries for Supported IDEs.

To find a target preferences block:

- Use the search feature in the Simulink Library Browser.
- Browse the block libraries for your processor or IDE.

You can identify a target preference block by its board icon and label. The label includes the processor name or “Custom Board”. For example:



## Configuring a Target Preferences Block for a Supported Processor

Before you can generate code for a model, your model must contain a target preferences (TP) block.

If you are using a supported processor, and a preconfigured TP block is not available from Target Support Package block libraries, configure a TP block for your processor.

To configure a TP block for a supported processor:

- 1 Open the block library for your IDE.
- 2 Copy the Custom Board block to your model.
- 3 Open the Custom Board block.

- 4 Select your target processor from the **Processor** parameter, verify the default settings, and click **OK**. This action imports the appropriate default settings and applies them to the model.
- 5 In your model, edit the label of the TP block with the name of your processor.

To make reusing the TP block easier:

- 1 In your model, select **File > New > Library**.
- 2 Copy your new TP block to the library.
- 3 Save the library in your default Current Folder in MATLAB.

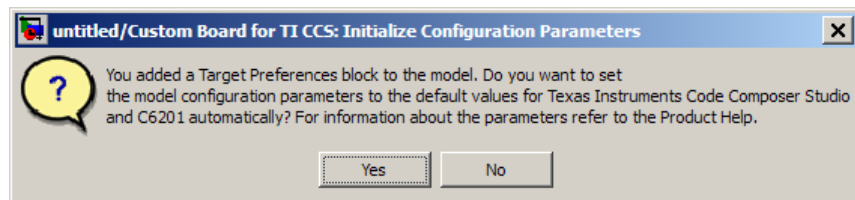
When you need the block again, open the library by entering the library name on the MATLAB command line.

### Adding a Target Preferences Block to Your Model

Before you can generate code for a model, your model must contain a target preferences (TP) block.

To add a TP block to your model:

- 1 Copy a TP block from a block library to your model, or create one, as described in “Configuring a Target Preferences Block for a Supported Processor” on page 2-3.
- 2 Click **Yes** if you get a dialog box that asks whether to “set the model configuration parameters to the default values”. For example:



This action applies the appropriate default settings for your IDE and processor to the Configuration Parameters dialog box.

Clicking **No** dismisses the dialog box and does not set the parameters. If the configuration parameters are incorrect, the software will generate error messages when you generate code. For more information, see “Setting Configuration Parameters for Embedded IDE Link” on page 2-6.

- 3 Open the TP block, verify the default settings, and click **OK**. This action applies the appropriate default settings to the model.

---

**Note** Your model must contain only one TP block.

---

Other tips for using TP blocks:

- The TP block stands alone. It does not connect to other blocks.
- To generate code for a model, place the TP block at the top level of your model.
- To generate code for a subsystem, place the TP block at the subsystem level of your model.
- For detailed information about the TP block parameters, see Target Preferences/Custom Board.

## Examples of Configuring Target Preferences

There is no generic procedure for configuring a target preferences block. Setting the **Processor** parameter applies the appropriate default for a specific processor.

You typically reconfigure TP to achieve a specific purpose. For example:

- “Configuring a Target Preferences Block for a Supported Processor” on page 2-3
- “Generating Code for Any C64x+™ Processor or Board”

# Setting Configuration Parameters for Embedded IDE Link

### In this section...

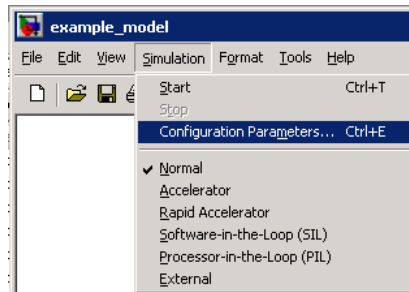
“What are Configuration Parameters?” on page 2-6

“Setting Model Configuration Parameters” on page 2-6

## What are Configuration Parameters?

The **Configuration Parameters** dialog box specifies the settings for a model's active *configuration set*. These parameters determine the type of solver used, import and export settings, and other values that determine how the model runs. See Configuration Sets for more information.

To display the dialog box, select **Simulation > Configuration Parameters** in the Model Editor, or press **Ctrl+E**. The dialog box appears.

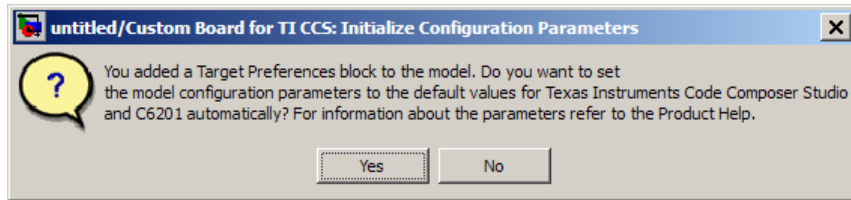


For comprehensive information about configuration parameters in Simulink see, “Configuration Parameters Dialog Box”

## Setting Model Configuration Parameters

The Embedded IDE Link software sets the appropriate default values for your processor and IDE when you drop a target preferences block in your model and click **Yes** in response to dialog box that asks whether to “set the model configuration parameters to the default values”. For example:





The following subsections provides a quick overview of the panes and parameters you are most with which you are most likely to interact.

refer to “About the TLC Debugger” in your Real-Time Workshop processor Language Compiler documentation.

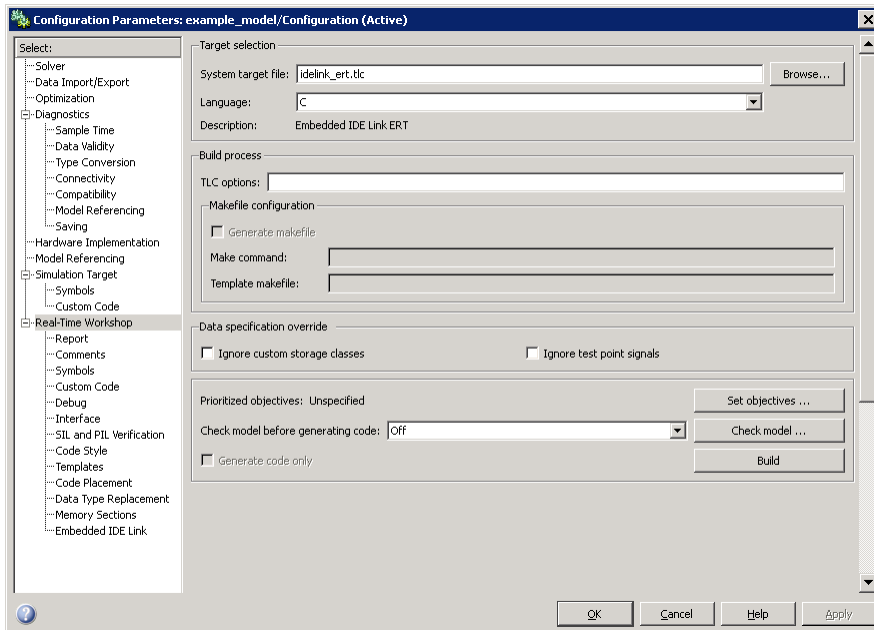
---

**Note** The following subtopics assume you’ve added a target preferences block to your model and accepted the default values.

---

### **Real-Time Workshop Pane**

The default **System target file** is `idelink_ert.tlc`. When you select `idelink_ert.tlc` or `idelink_grt.tlc`, the dialog box displays a new pane for Embedded IDE Link at the bottom of the select tree.



To use Real-Time Workshop Embedded Coder software or the Processor-in-the-Loop feature, leave **System target file** set to `idmlink_ert.tlc`.

Disregard the **Build process** options. Embedded IDE Link software does not use makefiles to generate code. Code generation is project based so the options in this group do not apply.

---

**Note** Embedded IDE Link software has a separate feature that automatically generates makefiles, which you can use to build applications with your software development toolchain. For more information, see [Generating Makefiles](#).

---

If you generate code from a model that uses custom storage classes (CSC), leave **Ignore custom storage classes** unselected.

To use a system target file that does not support CSCs, such as `idelink_grt.tlc`, without reconfiguring your parameter and signal objects, select **Ignore custom storage classes**. When you select **Ignore custom storage classes**:

- The software treats objects with CSCs as if you set their storage class attribute to Auto.
- The storage class of signals that have CSCs does not appear on the signal line, even when you select **Storage class** from **Format > Port/Signals Display** in your Simulink menus.

## Embedded IDE Link Pane Parameters

On the select tree, the Embedded IDE Link entry provides options in these areas:

- **Run-Time** — Set options for run-time operations, like the build action
- **Project Options** — Set build options for your project code generation
- **Code Generation** — Configure your code generation requirements
- **Link Automation** — Export an IDE handle object, such as `IDE_Obj`, to your MATLAB workspace
- **Diagnostic options** — Determine how the code generation process responds when you use source code replacement, either in the Target Preferences block **Board custom code** options, or in the Real-Time Workshop **Custom Code** options in the configuration parameters.

For more information, see *Embedded IDE Link Users Guide* .

**Build format.** Select **Project** to build a project for your IDE, or select **Makefile** to generate a makefile for your development tool chain.

For more information, see *Build Format*.

**Build action.** Your selection for **Build action** determines what happens when you click **Build** or press **Ctrl+B**. Your selection tells Real-Time Workshop software when to stop the code generation and build process.

To run your model on the processor, select **Build\_and\_execute**. This selection is the default build action; Real-Time Workshop software automatically downloads and runs the model on your board.

The actions are cumulative—each listed action adds features to the previous action on the list and includes all the previous features.

If you set **Build format** to **Project**, select one of the following options:

- **Create\_Project** — Directs Real-Time Workshop software to start the IDE and populate a new project with the files from the build process. This option offers a convenient way to build projects in the IDE.
- **Archive\_library** — Directs Real-Time Workshop software to archive the project for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your the IDE projects for models that you use in model referencing.
- **Build** — Builds the executable COFF file, but does not download the file to the processor.
- **Build\_and\_execute** — Directs Real-Time Workshop software to build, download, and run your generated code as an executable on your processor.
- **Create\_processor\_in\_the\_loop\_project** — Directs the Real-Time Workshop 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.
- **Archive\_library** — Creates a makefile and an archive library.
- **Build** — Creates a makefile and an executable.
- **Build\_and\_execute** — Creates a makefile and an executable. Then it evaluates the execute instruction in the current configuration. For more information, see **Execute**.

---

**Note** When you build and execute a model on your processor, the Real-Time Workshop software build process resets the processor automatically. You do not need to reset the board before building models.

---

For more information, see [Build action](#).

**Overrun notification.** To enable the overrun indicator, choose one of three ways for the processor to respond to an overrun condition in your model:

- **None** — Ignore overruns encountered while running the model.
- **Print\_message** — When the DSP encounters an overrun condition, it prints a message to the standard output device, `stdout`.
- **Call\_custom\_function** — Respond to overrun conditions by calling the custom function you identify in **Function name**.

For more information, see “Overrun notification” on page 10-10.

**Function name.** When you select `Call_custom_function` from the **Overrun notification** list, you enable this option. Enter the name of the function the processor should use to notify you that an overrun condition occurred. The function must exist in your code on the processor.

For more information, see “Function name” on page 10-12.

**Configuration.** The **Configuration** parameter defines sets of build options that apply to all of the files generated from your model.

The **Release** and **Debug** option apply build settings that are defined by your IDE. For more information, refer to your IDE documentation.

**Custom** has the same default values as **Release**, but:

- Leaves **Compiler options string** empty and `s`
- Specifies a memory model that uses `Far Aggregate` for data and `Far` for functions.

For more information, see “Configuration” on page 10-15.

**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, Embedded IDE Link does not set any optimization flags.

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**.

For more information, see “Compiler options string” on page 10-17.

**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, Embedded IDE Link does not set any linker options.

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**.

For more information, see “Linker options string” on page 10-19.

**System stack size (MAUs).** Enter the amount of memory to use for the stack. For more information, refer to **Enable local block outputs** on the **Optimization** pane of the Configuration Parameters dialog box. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory. Also refer to the online Help system for more information about Real-Time Workshop options for configuring and building models and generating code.

For more information, see “System stack size (MAUs)” on page 10-21.

**System heap size (MAUs).** Enter the amount of memory to use for the heap.

For more information, see “System heap size (MAUs)” on page 10-23.

**Profile real-time execution.** To enable the real-time execution profile capability, select **Profile real-time execution**. With this selected, the build process instruments your code to provide performance profiling at the task level or for atomic subsystems. When you run your code, the executed code reports the profiling information in an HTML report.

For more information, see “Profile real-time execution” on page 10-24.

**Link Automation.** When you use Real-Time Workshop to build a model for a processor, Embedded IDE Link makes a connection between MATLAB software and the IDE.

Constructors create objects that reference the link between the IDE and MATLAB. Link automation refers to the same object, named `IDE_Obj` in the function reference pages.

Although `IDE_Obj` is a bridge to a specific instance of the IDE, it is an object that contains information about the IDE instance it refers to, such as the board and processor it accesses. In this pane, the **Export IDE link handle to base workspace** option lets you instruct Embedded IDE Link to export the object to your MATLAB workspace, giving it the name you assign in **IDE link handle name**.

**Maximum time allowed to build project (s).** Specifies how long the software waits for the IDE to build the software.

For more information, see “Maximum time allowed to build project (s)” on page 10-30.

**Maximum time allowed to complete IDE operations (s).** Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages.

For more information, see “Maximum time allowed to complete IDE operations (s)” on page 10-32.

**Export IDE link handle to base workspace.** Directs the software to export the `IDE_Obj` object to your MATLAB workspace.

For more information, see “Export IDE link handle to base workspace” on page 10-33.

**IDE link handle name.** Specifies the name of the IDE\_Obj object that the build process creates.

For more information, see “IDE link handle name” on page 10-35.

**Source file replacement.** Selects the diagnostic action to take if the software detects conflicts when you replace source code with custom code. The diagnostic message responds to both source file replacement in the Embedded IDE Link parameters and in the Real-Time Workshop **Custom code** parameters in the configuration parameters for your model.

The following settings define the messages you see and how the code generation process responds:

- **none** — Does not generate warnings or errors when it finds conflicts.
- **warning** — Displays a warning. `warn` is the default value.
- **error** — Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

The build operation continues if you select `warning` and the software detects custom code replacement problems. 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. Use `none` when the replacement process is correct and you do not want to see multiple messages during your build.

For more information, see “Source file replacement” on page 10-36.



## **Working with Block Libraries**

For general information about working with block libraries in Simulink, see “Working with Block Libraries”.

# Simulink Models and Targeting

In this section...
“Creating Your Simulink Model for Targeting” on page 2-16
“Blocks to Avoid in Your Models” on page 2-17

## Creating Your Simulink Model for Targeting

You create real-time digital signal processing models the same way you create other Simulink models—by combining standard DSP blocks and C-MEX S-functions.

You add blocks to your model in several ways:

- Use blocks from the Signal Processing Blockset™ software
- Use other Simulink discrete-time blocks
- Use the blocks provided for your processor family
- Use blocks that provide the functions you need from any blockset installed on your computer
- Create and use custom blocks

Once you have designed and built your model, you generate C code and build the real-time executable by clicking **Build** on the **Real-Time Workshop** pane of the Configuration Parameters dialog box. The automatic build process creates the file `modelName.out` containing a real-time model image in COFF file format that can run on your target.

The file `modelName.out` is an executable whose format is target-specific. You can load the file to your target and execute it in real time. Refer to your Real-Time Workshop documentation for more information about the build process.

## Blocks to Avoid in Your Models

Many blocks in the blocksets communicate with your MATLAB workspace. All blocks generate code, but they do not work in the generated code as they do on your desktop.

You avoid using certain blocks, such as the Scope block and some source and sink blocks, in Simulink models that you use on Target Support Package targets. These blocks waste time in the generated code waiting to send or receive data from your MATLAB workspace, slowing your signal processing application without adding instrumentation value.

The following table describes blocks you should *not* use in your target models.

Block Name/Category	Library	Description
Scope	Simulink, Signal Processing Blockset software	Provides oscilloscope view of your output. Do not use the <b>Save data to workspace</b> option on the <b>Data history</b> pane in the Scope Parameters dialog box.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	Signal Processing Blockset	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	Signal Processing Blockset	Send data to your MATLAB workspace.

<b>Block Name/Category</b>	<b>Library</b>	<b>Description</b>
Signal To Workspace	Signal Processing Blockset	Send a signal to your MATLAB workspace.
Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	Signal Processing Blockset	Get a signal from your MATLAB workspace.
To Wave device	Signal Processing Blockset	Send data to a .wav device.
From Wave device	Signal Processing Blockset	Get data from a .wav device.

In general, using blocks to add instrumentation to your application is a valuable tool. In most cases, blocks you add to your model to display results or create plots, such as Histogram blocks, add to your generated code without affecting your running application.

# Generating IDE Projects

---

- “Introducing Project Generator” on page 3-2
- “Project Generation” on page 3-3
- “Schedulers and Timing” on page 3-4
- “Project Generator Tutorial” on page 3-23
- “Setting Code Generation Parameters for Processors” on page 3-29
- “Using Custom Source Files in Generated Projects” on page 3-32
- “Optimizing Embedded Code with Target Function Libraries” on page 3-36
- “Model Reference” on page 3-43

# Introducing Project Generator

Project generator provides the following features for developing projects and generating code:

- Support automated project building for IDE software that lets you create projects from code generated by Real-Time Workshop and Real-Time Workshop Embedded Coder products. The project automatically populates IDE projects in the IDE development environment.
- Configure code generation using model configuration parameters and processor preferences block options
- Select from two system target files to generate code specific to your processor
- Configure project build process
- Automatically download and run your generated projects on your processor

## Project Generation

Project Generator uses handle objects to connect to IDEs. Each time you build a model to generate a project, the build process uses one of the following constructors to create an IDE handle object:

- `altiumtasking` for Altium TASKING
- `adivdsp` for Analog Devices VisualDSP++
- `eclipseide` for Eclipse IDE
- `ghsmulti` for Green Hills MULTI
- `ticcs` for Texas Instruments' Code Composer Studio

The software attempts to connect to the board (`boardnum`) and processor (`procnum`) associated with the **Board name** and **Processor number** parameters in the Target Preferences block in the model.

## Schedulers and Timing

In this section...
“Configuring Models for Asynchronous Scheduling” on page 3-4
“Cases for Using Asynchronous Scheduling” on page 3-5
“Using Scheduling Blocks to Control Code Execution” on page 3-7
“Comparing Synchronous and Asynchronous Interrupt Processing” on page 3-7
“Using Synchronous Scheduling” on page 3-9
“Using Asynchronous Scheduling” on page 3-10
“Multitasking Scheduler Examples” on page 3-10

### Configuring Models for Asynchronous Scheduling

Using the scheduling blocks, you can use an asynchronous (real-time) scheduler for your processor application. The asynchronous scheduler enables you to define interrupts and tasks to occur when you want by using blocks in the following block libraries:

- `idelinklib_common`

---

#### Note

- One way to view the block libraries is by entering the block library name at the MATLAB command line. For example: `>> idelinklib_common`
  - You cannot build and run the models in following examples without additional blocks. They are for illustrative purposes only.
- 

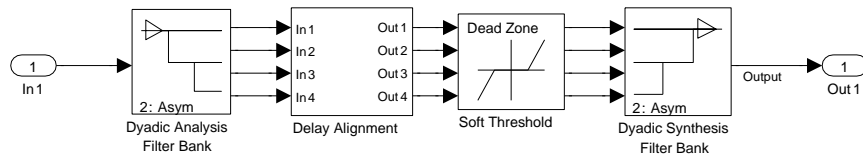
Also, you can schedule multiple tasks for asynchronous execution using the blocks.



The following figures show a model updated to use the asynchronous scheduler by converting the model to a function subsystem and then adding a scheduling block (Hardware Interrupt) to drive the function subsystem in response to interrupts.

### Before

The following model uses synchronous scheduling provided by the base rate in the model.

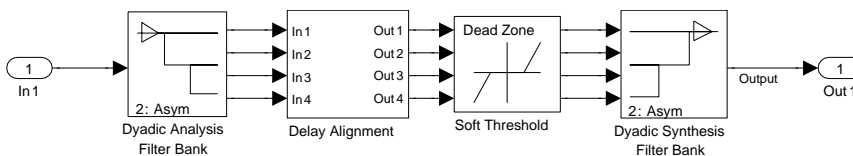


### After

To convert to asynchronous operation, wrap the model in the previous figure in a function block and drive the input from a Hardware Interrupt block. The hardware interrupts that trigger the Hardware Interrupt block to activate an ISR now triggers the model inside the function block.

### Algorithm Inside the Function Call Subsystem Block

Here's the model inside the function call subsystem in the previous figure. It is the same as the original model that used synchronous scheduling.

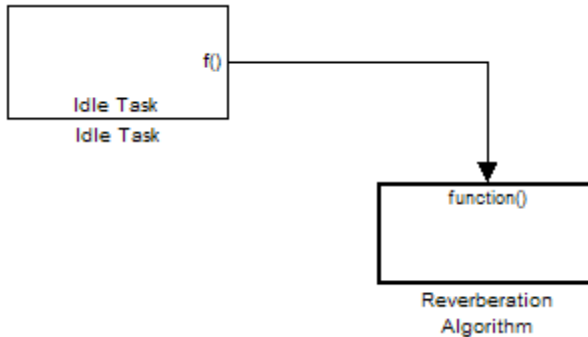


## Cases for Using Asynchronous Scheduling

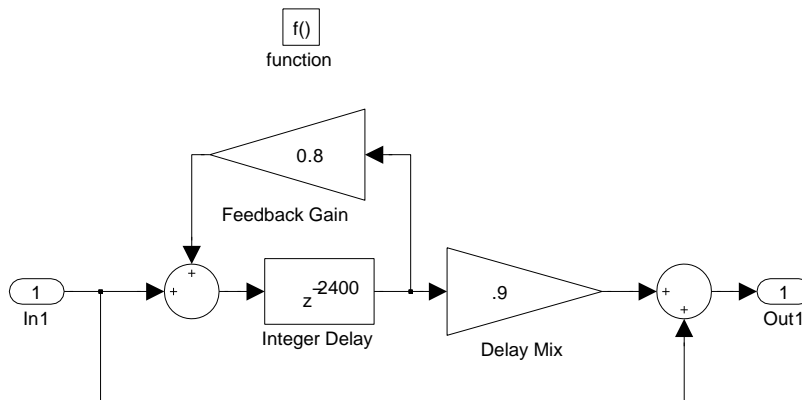
The following sections present common cases for using the scheduling blocks described in the previous sections.

### Idle Task

The following model illustrates a case where the reverberation algorithm runs in the context of a background task in bare-board code generation mode.



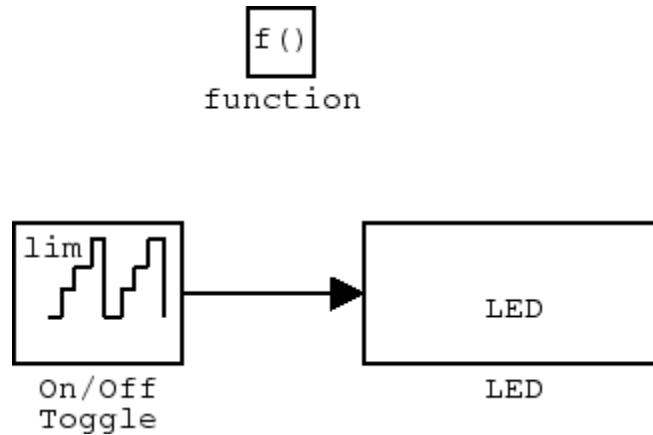
The function generated for this task normally runs in free-running mode—repetitively and indefinitely. Subsystem execution of the reverberation function is data driven via a background DMA interrupt-controlled ISR, shown in the following figure.



### Hardware Interrupt Triggered Task

In the next figure, you see a case where a function (LED Control) runs in the context of a hardware interrupt triggered task.

In this model, the Hardware Interrupt block installs a task that runs when it detects an external interrupt. This task performs the specified function with an LED.



## Using Scheduling Blocks to Control Code Execution

Embedded IDE Link Hardware Interrupt blocks enable selected hardware interrupts processors, generate corresponding ISRs, and connect them to the corresponding interrupt service vector table entries.

When you connect the output of the Hardware Interrupt block to the control input of a function-call subsystem, the generated subsystem code is called from the ISRs each time the interrupt is raised.

The Idle Task block specifies one or more functions to execute as background tasks in the code generated for the model. The functions are created from the function-call subsystems to which the Idle Task block is connected. For more information, see “Idle Task” on page 3-6.

## Comparing Synchronous and Asynchronous Interrupt Processing

Code generated for periodic tasks, both single- and multitasking, runs via a timer interrupt. A timer interrupt ensures that the generated code representing periodic-task model blocks runs at the specified period. The

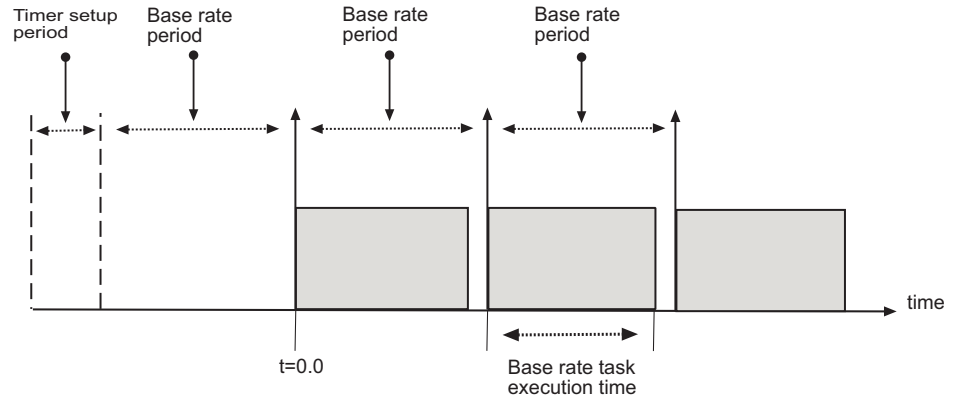
periodic interrupt clocks code execution at runtime. This periodic interrupt clock operates on a period equal to the base sample time of your model.

---

**Note** The execution of synchronous tasks in the model commences at the time of the first timer interrupt. Such interrupt occurs at the end of one full base rate period which follows timer setup. The time of the start of the execution corresponds to  $t=0$ .

---

The following figure shows the relationship between model startup and execution. Execution starts where your model executes the first interrupt, offset to the right of  $t=0$  from the beginning of the time line. Before the first interrupt, the simulation goes through the timer set up period and one base rate period.



Timer-based scheduling does not provide enough flexibility for some systems. Systems for control and communications must respond to asynchronous events in real time. Such systems may need to handle a variety of hardware interrupts in an asynchronous, or *aperiodic*, fashion.

When you plan your project or algorithm, select your scheduling technique based on your application needs.

- If your application processes hardware interrupts asynchronously, add the appropriate asynchronous scheduling blocks from the library to your model:

- A Hardware Interrupt block, to create an interrupt service routine to handle hardware interrupts on the selected processor
- An Idle Task block, to create a task that runs as a separate thread
- Simulink sets the base rate priority to 40, the lowest priority.
- If your application does not service asynchronous interrupts, include only the algorithm and device driver blocks that specify the periodic sample times in the model.

---

**Note** Generating code from a model that does not service asynchronous interrupts automatically enables and manages a timer interrupt. The periodic timer interrupt clocks the entire model.

---

## Using Synchronous Scheduling

Code that runs synchronously via a timer interrupt requires an interrupt service routine (ISR). Each model iteration runs after an ISR services a posted interrupt. The code generated for Embedded IDE Link uses a timer. To calculate the timer period, the software uses the following equation:

$$Timer\_Period = \frac{(CPU\_Clock\_Rate) * (Base\_Sample\_Time)}{Low\_Resolution\_Clock\_Divider} * Prescaler$$

The software configures the timer so that the base rate sample time for the coded process corresponds to the interrupt rate. Embedded IDE Link calculates and configures the timer period to ensure the desired sample rate.

Different processor families use the timer resource and interrupt number differently. Entries in the following table show the resources each family uses.

The minimum base rate sample time you can achieve depends on two factors—the algorithm complexity and the CPU clock speed. The maximum value depends on the maximum timer period value and the CPU clock speed.

If all the blocks in the model inherit their sample time value, and you do not define the sample time, Simulink assigns a default sample time of 0.2 second.

### Using Asynchronous Scheduling

Embedded IDE Link enables you to model and automatically generate code for asynchronous systems. To do so, use the following scheduling blocks:

- Hardware Interrupt (for bare-board code generation mode)
- Idle Task

The Hardware Interrupt block operates by

- Enabling selected hardware interrupts for the processor
- Generating corresponding ISRs for the interrupts
- Connecting the ISRs to the corresponding interrupt service vector table entries

---

**Note** You are responsible for mapping and enabling the interrupts you specify in the block dialog box.

---

Connect the output of the Hardware Interrupt block to the control input of a function-call subsystem. By doing so, you enable the ISRs to call the generated subsystem code each time the hardware raises the interrupt.

The Idle Task block specifies one or more functions to execute as background tasks in the code generated for the model. The functions are created from the function-call subsystems to which the Idle Task block is connected.

### Multitasking Scheduler Examples

Embedded IDE Link provides a scheduler that supports multiple tasks running concurrently and preemption between tasks running at the same time. The ability to preempt running tasks enables a wide range of scheduling configurations.

Multitasking scheduling also means that overruns, where a task runs beyond its intended time, can occur during execution.

To understand these examples, you must be familiar with the following scheduling concepts:

- *Preemption* is the ability of one task to pause the processing of a running task to run instead. With the multitasking scheduler, you can define a task as preemptible—thus, another task can pause (preempt) the task that allows preemption. The scheduler examples in this section that demonstrate preemption, illustrate one or more tasks allowing preemption.
- *Overrunning* occurs when a task does not reach completion before it is scheduled to run again. For example, overrunning can occur when a Base-Rate task does not finish in 1 ms. Overrunning delays the next execution of the overrunning task and may delay execution of other tasks.

Examples in this section demonstrate a variety of multitasking configurations:

- “Three Odd-Rate Tasks Without Preemption and Overruns” on page 3-13
- “Two Tasks with the Base-Rate Task Overrunning, No Preemption” on page 3-14
- “Two Tasks with Sub-Rate 1 Overrunning Without Preemption” on page 3-15
- “Three Even-Rate Tasks with Preemption and No Overruns” on page 3-16
- “Three Odd-Rate Tasks Without Preemption and the Base and Sub-Rate 1 Tasks Overrun” on page 3-18
- “Three Odd-Rate Tasks with Preemption and Sub-Rate 1 Task Overruns” on page 3-19
- “Three Even-Rate Tasks with Preemption and the Base-Rate and Sub-Rate 1 Tasks Overrun” on page 3-21

Each example presents either two or three tasks:

- **Base Rate task.** Base rate is the highest rate in the model or application. The examples use a base rate of 1ms so that the task should execute every one millisecond.
- **Sub-Rate 1.** The first subrate task. Sub-Rate 1 task runs more slowly than the Base-Rate task. Sub-Rate 1 task rate is 2ms in the examples so that the task should execute every 2ms.




- **Sub-Rate 2.** In examples with three tasks, the second subrate task is called Sub-Rate 2. Sub-Rate 2 tasks run more slowly than Sub-Rate 1. In the examples, Sub-Rate 2 runs at either 4ms or 3ms.
  - When Sub-Rate 2 is 4ms, the example is called *even*.
  - When Sub-Rate 2 is 3ms, the example is called *odd*.

---

**Note** The odd or even naming only identifies Sub-Rate 2 as being 3 or 4ms. It does not affect or predict the performance of the tasks.

---

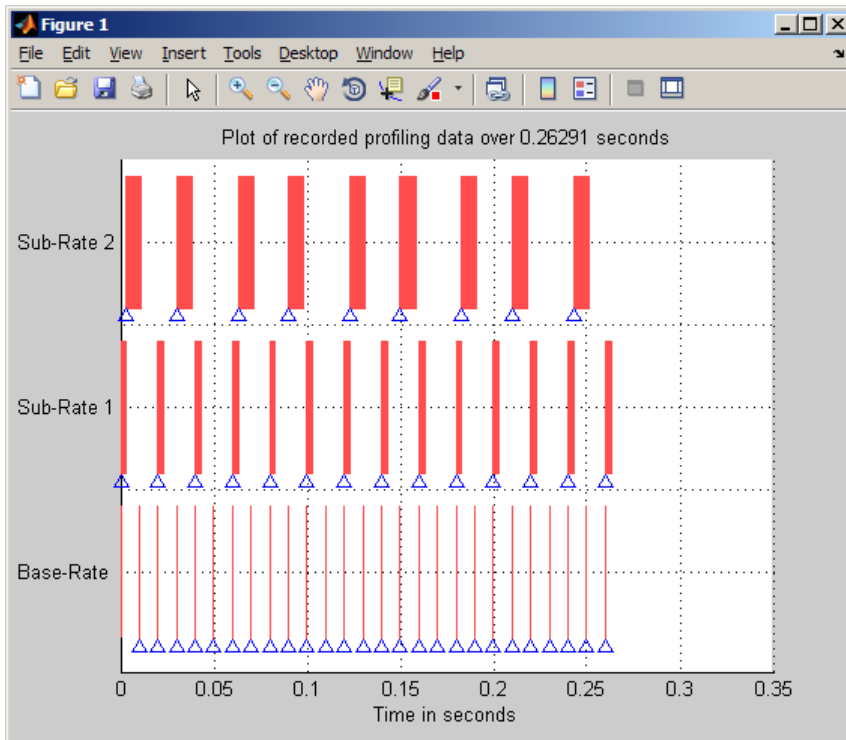
The following legend applies to the plots in the next sections:

- Blue triangles (  ) indicate when the task started.
- Dark red areas (  ) indicate the period during which a task is running
- Pink areas (  ) within dark red areas indicate a period during which a running task is suspended—preempted by a task with higher priority



### Three Odd-Rate Tasks Without Preemption and Overruns

In this three task scenario, all of the tasks run as scheduled. No overruns or preemptions occur.

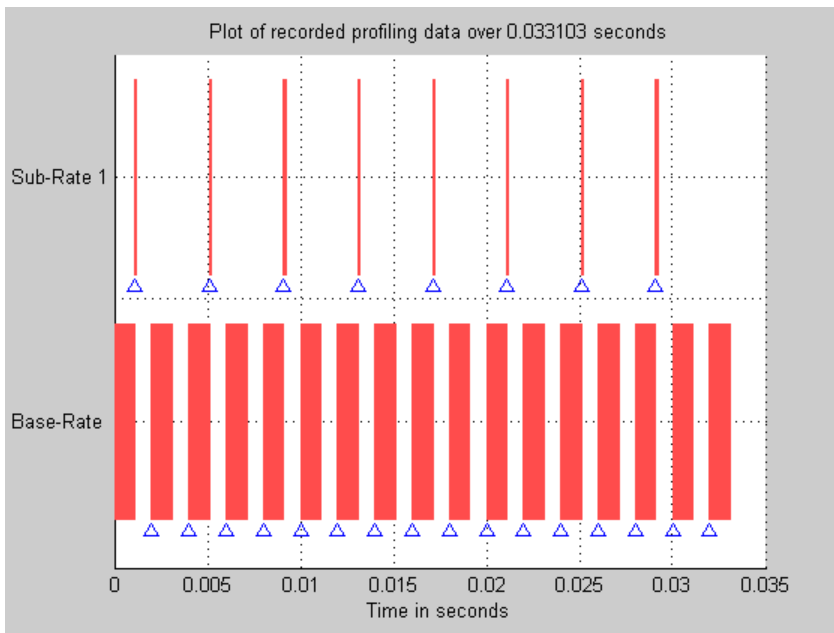


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	1ms
Sub-Rate 1	2ms	2ms
Sub-Rate 2	3ms	3ms

### Two Tasks with the Base-Rate Task Overrunning, No Preemption

In this two rate scenario, the Base-Rate overruns the 1ms time intended and prevents the subrate task from completing successfully or running every 2ms.

- Sub-Rate 1 does not allow preemption and fails to run when scheduled, but is never interrupted.
- The Base-Rate runs every 2ms and Sub-Rate 1 runs every 4ms instead of 2ms.

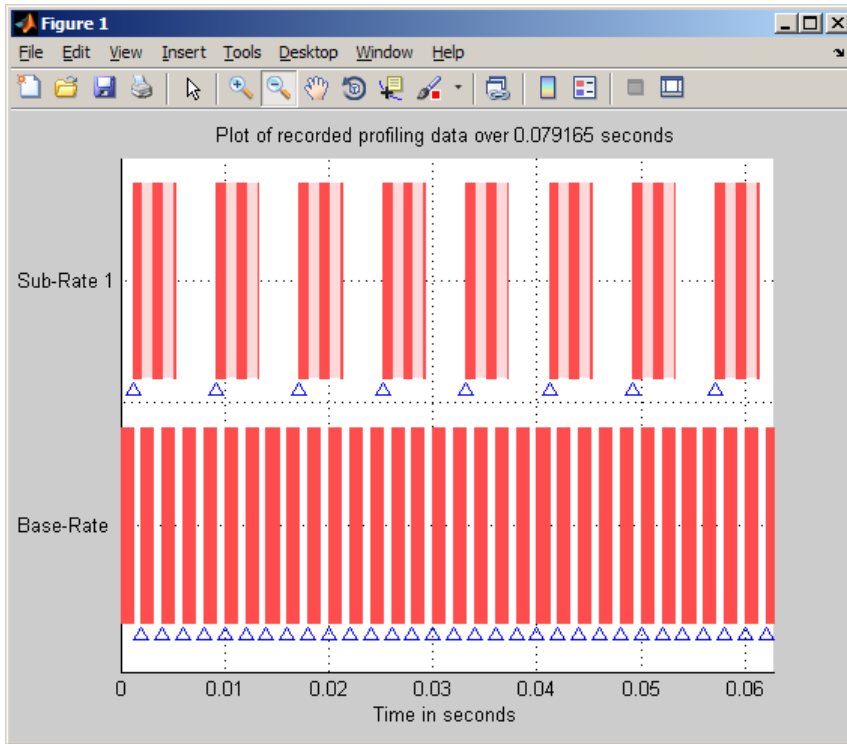


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)

## Two Tasks with Sub-Rate 1 Overrunning Without Preemption

In this example, two rates running simultaneously—the Base-Rate task and one subrate task. Both the Base-Rate task and the Sub-Rate 1 task overrun.

- Base-Rate runs every 2ms instead of 1ms.
  - The Sub-Rate 1 task both overruns and is affected by the Base-Rate task overrunning.
  - The Base-Rate task overrun delays Sub-Rate 1 task execution by a factor of 4.
- Sub-Rate 1 runs every 8ms rather than every 2ms.
- The Base-Rate runs at 1ms.
- The Base-Rate task preempts Sub-Rate 1 when it tries to execute.
- The Sub-Rate 1 tasks overrun, taking up to 5ms to complete rather than 2ms.

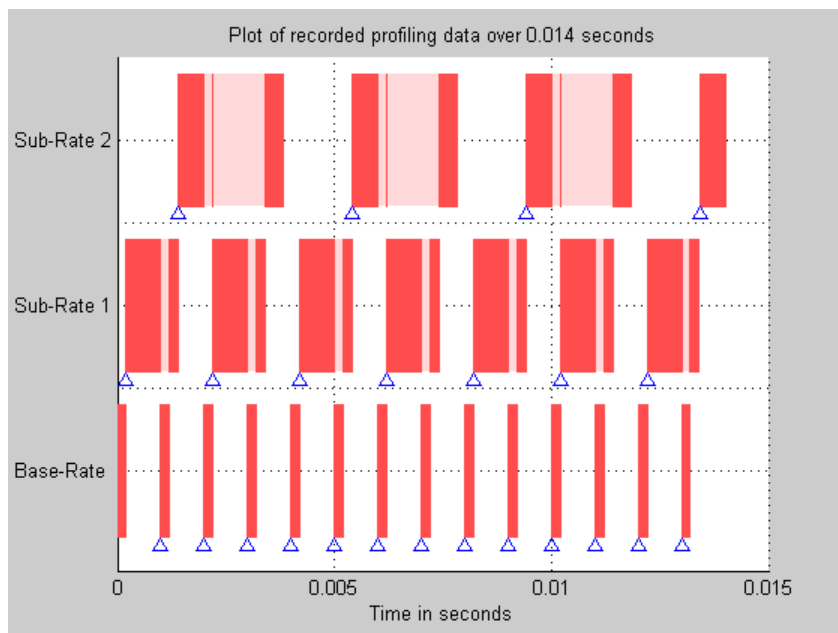


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	8ms (overrunning)

### Three Even-Rate Tasks with Preemption and No Overruns

In the following three task scenario, the Base-Rate runs as scheduled and preempts Sub-Rate 1.

- Both the Base-Rate and Sub-Rate 1 tasks preempt Sub-Rate 2 task execution.
- Preempting the subrate tasks does not prevent the subrate tasks from running on schedule.

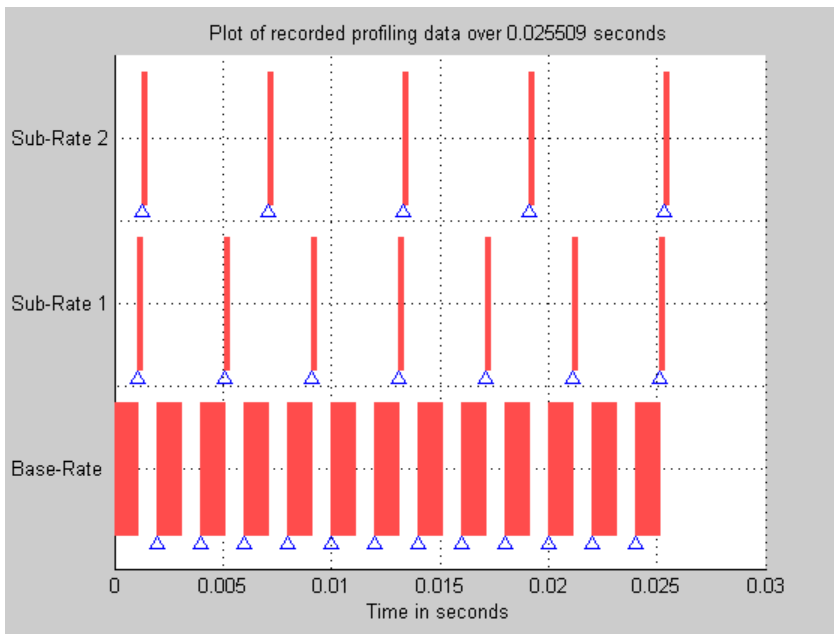


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	1ms
Sub-Rate 1	2ms	2ms
Sub-Rate 2	4ms	4ms

### Three Odd-Rate Tasks Without Preemption and the Base and Sub-Rate1 Tasks Overrun

Three tasks running simultaneously—the Base-Rate task and two subrate tasks.

- Both the Base-Rate task and the Sub-Rate 1 task overrun.
- The Base-Rate task runs every 2ms instead of 1ms.
- Sub-Rate 1 and Sub-Rate 2 task execution is delayed by a factor of 2—Sub-Rate 1 runs every 4ms rather than every 2ms and Sub-Rate 2 runs every 6ms instead of 3ms.

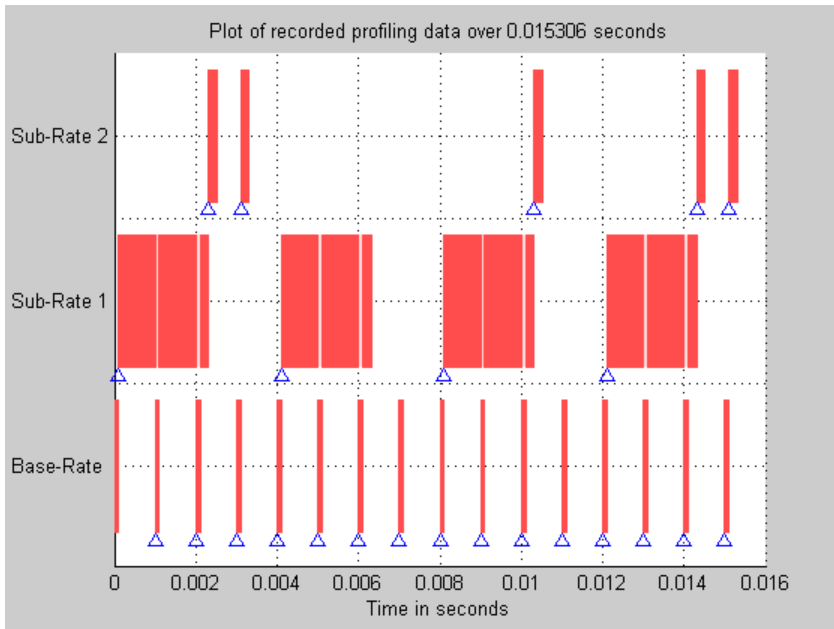


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)
Sub-Rate 2	3ms	6ms (overrunning)

### Three Odd-Rate Tasks with Preemption and Sub-Rate 1 Task Overruns

In this three task scenario, the Base-Rate preempts Sub-Rate 1 which is overrunning.

- The overrunning subrate causes Sub-Rate 1 to execute every 4ms instead of 2ms.
- Every other fourth execution of Sub-Rate 2 does not occur.
- Instead of executing at  $t=0, 3, 6, 9, 12, 15, 18, \dots$ , Sub-Rate 2 executes at  $t=0, 3, 9, 12, 15, 21$ , and so on.
- The  $t=6$  and  $t=18$  instances do not occur.



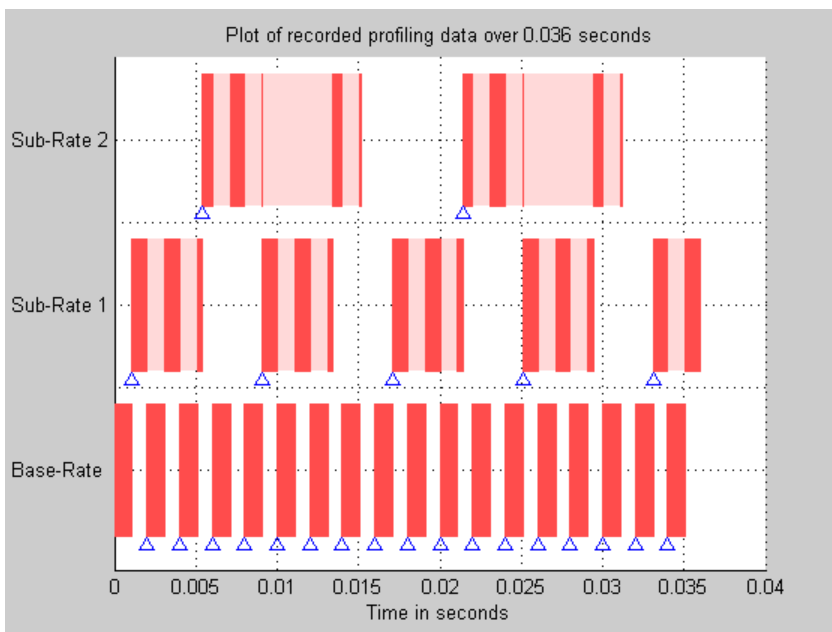
<b>Task Identification</b>	<b>Intended Execution Schedule</b>	<b>Actual Execution Schedule</b>
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)
Sub-Rate 2	3ms	6ms (overrunning and skipping every other fourth execution)



### Three Even-Rate Tasks with Preemption and the Base-Rate and Sub-Rate 1 Tasks Overrun

In this three-task scenario, two of the tasks overrun—the Base-Rate and Sub-Rate 1.

- The overrunning Base-Rate executes every 2ms.
- Sub-Rate 1 overruns due to the Base-Rate overrun, doubling the execution rate.
- Also, Sub-Rate 1 is overrunning as well, doubling the execution rate again, from the intended 2ms to 8ms.
- Sub-Rate 2 responds to the overrunning Base-Rate and Sub-Rate 1 tasks by running every 16ms instead of every 4ms.



<b>Task Identification</b>	<b>Intended Execution Schedule</b>	<b>Actual Execution Schedule</b>
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	8ms (overrunning)
Sub-Rate 2	3ms	16ms (overrunning)

# Project Generator Tutorial

In this section...
“Creating the Model” on page 3-24
“Adding the Target Preferences Block to Your Model” on page 3-24
“Specify Configuration Parameters for Your Model” on page 3-26

In this tutorial you will use the Embedded IDE Link software to:

- Build a model.
- Generate a project from the model.
- Build the project and run the binary on a processor.

---

**Note** The model demonstrates project generation. You cannot not build and run the model on your processor without additional blocks.

---

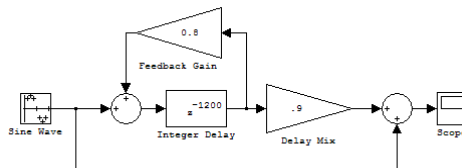
To generate a project from a model, complete the following tasks:

- 1** Create a model application.
- 2** Add a Target Preferences block from the Embedded IDE Link library to your model.
- 3** In the Target Preferences block, verify and set the block parameters for your hardware or simulator.
- 4** Set the configuration parameters for your model, including
  - Solver parameters such as simulation start and solver options
  - Real-Time Workshop software options such as processor configuration and processor compiler selection
- 5** Generate your project.
- 6** Review your project in the IDE.

## Creating the Model

To create the model for audio reverberation, follow these steps:

- 1 Start Simulink software.
- 2 Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.
- 3 Use Simulink blocks and Signal Processing Blockset blocks to create the following model.



Look for the Integer Delay block in the Discrete library of Simulink blocks and the Gain block in the Commonly Used Blocks library. Do not add the Custom Board block for the IDE at this time.

- 4 Save your model with a suitable name before continuing.

## Adding the Target Preferences Block to Your Model

To configure your model to work with a specific processor, use a Target Preferences/Custom Board block. These are available in the block library for your processor and in the following IDE block libraries:

- idelinklib\_avidvsp
- idelinklib\_ghsmulti
- idelinklib\_ticcs
- idelinklib\_eclipseide

Adding a Target Preferences block to a model triggers a dialog box that asks about your model configuration settings. The message tells you that the model configuration parameters will be set to default values based on the processor

specified in the block parameters. To set the parameters automatically, click **Yes**. Clicking **No** dismisses the dialog box and does not set the parameters.

When you click **Yes**, the software sets the system target file to `idelink_grt.tlc` or `idelink_ert.tlc` and sets the hardware options and product-specific parameters in the model to default values. If you open the model Configuration Parameters, you see the Embedded IDE Link pane option on the select tree.

Clicking **No** prevents the software from setting the system target file and the product specific options. When you open the model Configuration Parameters for your model, you do not see the Embedded IDE Link pane option on the select tree. To enable the options, select the `idelink_ert.tlc` or `idelink_grt.tlc` system target file from the System Target File list in the Real-Time Workshop pane options.

To add the target preferences block to your model, follow these steps:

- 1** Open the block library for your IDE or processor.
- 2** Drag and drop the target preferences block to your model.
- 3** Open the target preferences block by double-clicking it.
- 4** In the target preferences block dialog box, select your processor from the **Processor** list.
- 5** If Verify the **CPU clock** value and, if you are using a simulator, select **Simulator**.
- 6** Verify the settings on the **Memory** and **Sections** tabs to be sure they are correct for the processor you selected.
- 7** Click **OK** to close the Target Preferences dialog box.

You have completed the model. Now configure the model configuration parameters to generate a project in the IDE from your model.

---

**Note** To configure your model to run on Windows or Linux, see Preparing Models to Run on Windows or Linux.

---

### Specify Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog box in Simulink software.

#### Setting Solver Parameters

After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

- 1 Open the Configuration Parameters dialog box and set the appropriate options on the **Solver** category for your model and for Embedded IDE Link.
  - Set **Start time** to 0.0 and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to `inf` for completeness.
  - Under **Solver options**, select the **fixed-step** and **discrete** settings from the lists
  - Set the **Fixed step size** to **Auto** and the **Tasking Mode** to **Single Tasking**

---

**Note** Generated code does not honor Simulink software stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, add a Stop Simulation block in your model.

---

When you use PIL, you can set the **Solver options** to any selection from the **Type** and **Solver** lists.

Ignore the **Data Import/Export**, **Diagnostics**, and **Optimization** categories in the **Configuration Parameters** dialog box. The default settings are correct for your new model.

### Setting Real-Time Workshop Code Generation Parameters

To configure Real-Time Workshop software to use the correct processor files and to compile and run your model executable file, set the options in the **Real-Time Workshop** category of the **Select** tree in the **Configuration Parameters** dialog box. Follow these steps to set the code generation options for your DSP:

- 1 Select **Real-Time Workshop** on the **Select** tree.
- 2 In **Target** selection, use the **Browse** button to set **System target file** to `idelink_grt.tlc`.

### Setting Embedded IDE Link Parameters

To configure Real-Time Workshop software to use the correct code generation options and to compile and run your model executable file, set the options in the **Embedded IDE Link** category of the **Select** tree in the **Configuration Parameters** dialog box. Follow these steps to set the code generation options for your processor:

- 1 From the **Select** tree, choose **Embedded IDE Link** to specify code generation options that apply to your processor.
- 2 Set the following options in the pane under **Configuration**:
  - **Configuration** should be **Custom**.
  - Set **Compiler options string** and **Linker options string** should be blank.
- 3 Under **Link Automation**, verify that **Export IDE link handle to base workspace** is selected and provide a name for the handle in **IDE handle name** (optional).
- 4 Set the following **Runtime** options:
  - **Build action**: `Build_and_execute`.

- **Overrun notification:** None.

You have configured the Real-Time Workshop software options that let you generate a project for your processor. You may have noticed that you did not configure a few categories on the **Select** tree, such as **Comments**, **Symbols**, and **Optimization**.

For your new model, the default values for the options in these categories are correct. For other models you develop, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code. Refer to your Simulink and Real-Time Workshop documentation for more information about setting the configuration parameters.

### **Building Your Project**

After you set the configuration parameters and configure Real-Time Workshop software to create the files you need, you direct the build process to create your project:

- 1** Press **OK** to close the Configuration Parameters dialog box.
- 2** Click **Ctrl+B** to generate your project in the IDE.

When you click **Build** with `Create_project` selected for **Build action**, the automatic build process starts the IDE, populates a new project in the development environment, builds the project, loads the binary on the processor, and runs it.

- 3** To stop processor execution, use the **Halt** option in the IDE or enter `IDE_Obj.halt` at the MATLAB command prompt. (Where “`IDE_Obj`” is the IDE handle name you specified previously in **Configuration Parameters**.)



## Setting Code Generation Parameters for Processors

Before you generate code with Real-Time Workshop software, set the fixed-step solver step size and specify an appropriate fixed-step solver if the model contains any continuous-time states. At this time, you should also select an appropriate sample rate for your system. Refer to your *Real-Time Workshop User's Guide* documentation for additional information.

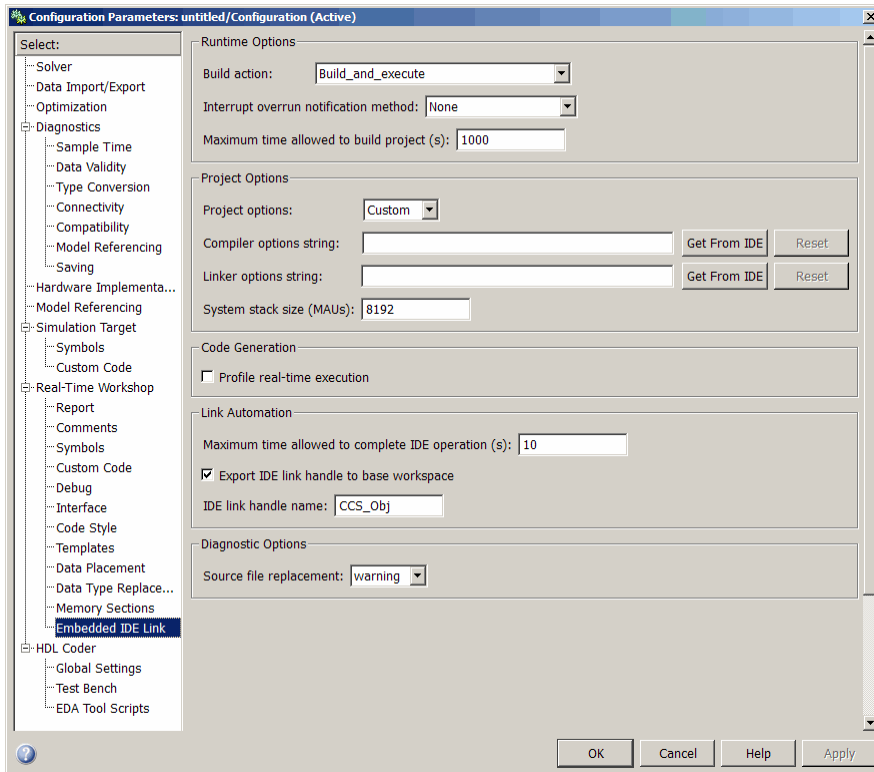
---

**Note** Embedded IDE Link does not support continuous states in Simulink software models for code generation. In the **Solver options** in the Configuration Parameters dialog box, you must select **Discrete (no continuous states)** as the **Type**, along with **Fixed step**.

---

The Real-Time Workshop pane of the Configuration Parameters dialog box lets you set numerous options for the real-time model. To open the Configuration Parameters dialog box, select **Simulation > Configuration Parameters** from the menu bar in your model.

The following figure shows the configuration parameters categories when you are using Embedded IDE Link.



In the **Select** tree, the categories provide access to the options you use to control how Real-Time Workshop software builds and runs your model. The first categories under **Real-Time Workshop** in the tree apply to all Real-Time Workshop software processors. They always appear on the list.

The last category under **Real-Time Workshop** is specific to the Embedded IDE Link system target files `sidelink_grt.tlc` and `idelink_ert.tlc` and appear when you select either file.

When you select your processor file in **Target Selection** on the **Real-Time Workshop** pane, the options change in the tree.

For Embedded IDE Link, the processor to select is `idelink_grt.tlc`. Selecting either the `idelink_grt.tlc` or `idelink_ert.tlc` adds the Embedded IDE Link options to the **Select** tree. The `idelink_grt.tlc` file is

appropriate for all projects. Select `idmlink_ert.tlc` when you are developing projects or code for embedded processors (requires Real-Time Workshop Embedded Coder software) or you plan to use Processor-in-the-Loop features.

The following sections present each configuration parameters **Select** tree category and the relevant options available in each.

## Using Custom Source Files in Generated Projects

In this section...
“Preparing to Replace Generated Files With Custom Files” on page 3-32
“Replacing Generated Source Files with Custom Files When You Generate Code” on page 3-34

The **Board custom code** options on the **Board Info** pane in the model's Target Preferences block enable you to replace a generated file with a custom file you provide. By replacing a file, you can modify the output of the code generation process to suit your needs. For file replacement during the code generation process to work, your custom file must have the same name as the file to replace.

The following sections show you how to:

- Identify the file to replace — “Preparing to Replace Generated Files With Custom Files” on page 3-32
- Create the custom replacement file — “Preparing to Replace Generated Files With Custom Files” on page 3-32
- Configure the target preferences to use the custom file when you generate a project — “Replacing Generated Source Files with Custom Files When You Generate Code” on page 3-34

For more information about the target preferences and the Board custom code options, refer to Target Preferences/Custom Board in the online Help system.

### Preparing to Replace Generated Files With Custom Files

To change the content of a generated project, use custom code replacement to replace a generated file in the project. By replacing a generated file in a project, you can make changes like the following in your generated project:

- Edit the file that contains linker directives to change the linking process, such as mapping memory differently.

- Modify a library file, such as a chip support library (.cs1 file)
- Add commands to a header file
- Modify data in a data file
- Add comments to a file for tracking or identifying the file or project

### Determining the Name of the File to Replace

To replace a file created when you generate a project, you need the name of the file to replace; the content to change; and the replacement file, including the path to the file. Your model must include a target preferences block configured for your processor, either a simulator or hardware.

The linker file for each IDE is has the same name as the model file, followed by a different extension. For example:

- For CCS, `modelName.cmd`.
- For MULTI, `modelName.ld`
- For VisualDSP++, `modelName.ldf`

Follow these steps to identify the file to replace in a project:

- 1** Open the configuration parameters for your model.
- 2** On the **Select** tree in the Configuration Parameter dialog box, select .
- 3** Set **Build action** to any entry on the list except **Create processor-in-the-loop project**.
- 4** Click **OK** to close the dialog box.
- 5** Press **Ctrl+B** to build your model.
- 6** Look at the files in the project in the IDE. Find the file that contains the information to supplement or replace.

The build process stores the project files in a directory named after your model and IDE, under in your MATLAB working directory. For example, `modelname_ccslink`, `modelname_multilink`, or `modelname_vdsplink`.

It also shows the project directory information in the MATLAB Command window.

- 7 Note the file name and location. You use this information to create your custom replacement file.

### Creating the Replacement File

To replace a file in a project during code generation, you need a new file with the same name saved in a different directory. Creating your replacement file from the file to replace increases the chances that the generated code will work properly with the new file. The new file must have all of the information the final project needs.

Follow these steps to create a file to use to replace a generated file in your project.

- 1 Determine the name of the file to replace. Refer to “Determining the Name of the File to Replace” on page 3-33 for how to do this.
- 2 Locate the file to replace. Copy the file and save it with the same name in a new directory.
- 3 Open your new file and edit the file to add or remove the information to change.
- 4 Save your changes to the file.

### Replacing Generated Source Files with Custom Files When You Generate Code

With the replacement file and location available, configure the build process to use your replacement file. Parameters on the Target Preferences block dialog box allow you to specify the replacement file to use. For more information about the board custom code options, refer to Target Preferences/Custom Board.

Follow these steps to configure the build process to use a replacement file.

- 1** Double-click **Target Preferences** in your model.
- 2** In the **Board** custom code options, select the type of file to replace—**Source files** or **Libraries**.
- 3** Enter the name of your replacement file and path in the text field.  
The build process recognizes two directory path tokens:
  - `$MATLAB` to refer to your MATLAB root directory
  - `$install_dir` to refer to the root of your IDE installation.
- 4** Click **OK** to close the dialog box.
- 5** Open the configuration parameters for your model and select the **Build action** to use to build your model.
- 6** From the **Source code replacement** list, select **warning** or **error** to see messages when the build process replaces files.
- 7** Click **OK** to save your configuration.
- 8** Return to the model window and press **Ctrl+B** to build your project. The generated project contains your replacement file instead of generating the matching file.

## Optimizing Embedded Code with Target Function Libraries

### In this section...

“About Target Function Libraries and Optimization” on page 3-36

“Using a Processor-Specific Target Function Library to Optimize Code” on page 3-38

“Process of Determining Optimization Effects Using Real-Time Profiling Capability” on page 3-39

“Reviewing Processor-Specific Target Function Library Changes in Generated Code” on page 3-40

“Reviewing Target Function Library Operators and Functions” on page 3-42

“Creating Your Own Target Function Library” on page 3-42

### About Target Function Libraries and Optimization

A *target function library* is a set of one or more function tables that define processor- and compiler-specific implementations of functions and arithmetic operators. The code generation process uses these tables when it generates code from your Simulink model.

The software registers processor-specific target function libraries during installation. To use one of the libraries, select the set of tables that correspond to functions implemented by intrinsics or assembly code for your processor from the **Target function library** list in the model configuration parameters. To do this, complete the following steps:

- 1 In your model, select **Simulation > Configuration Parameters**.
- 2 In the Configuration Parameters dialog box, select **Real-Time Workshop** and **Interface**.
- 3 Set the **Target function library** parameter to the appropriate library for your processor.

After you select the processor-specific library, the model build process uses the library contents to optimize generated code for that processor. The generated code includes processor-specific implementations for `sum`, `sub`, `mult`, and `div`,



and various functions, such as `tan` or `abs`, instead of the default ANSI<sup>®</sup> C instructions and functions. The optimized code enables your embedded application to run more efficiently and quickly, and in many cases, reduces the size of the code. For more information about target function libraries, refer to “Introduction to Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation.

### **Code Generation Using the Target Function Library**

The build process begins by converting your model and its configuration set to an intermediate form that reflects the blocks and configurations in the model. Then the code generation phase starts.

---

**Note** Real-Time Workshop refers to the following conversion process as replacement and it occurs before the build process generates a project.

---

During code generation for your model, the following process occurs:

- 1** Code generation encounters a call site for a function or arithmetic operator and creates and partially populates a target function library entry object.
- 2** The entry object queries the target function library database for an equivalent math function or operator. The information provided by the code generation process for the entry object includes the function or operator key, and the conceptual argument list.
- 3** The code generation process passes the target function library entry object to the target function library.
- 4** If there is a matching table entry in the target function library, the query returns a fully-populated target function library entry to the call site, including the implementation function name, argument list, and build information
- 5** The code generation process uses the returned information to generate code.

Within the target function library that you select for your model, the software searches the tables that comprise the library. The search occurs in the order in which the tables appear in either the Target Function Library Viewer or

the **Target function library** tool tip. For each table searched, if the search finds multiple matches for a target function library entry object, priority level determines the match to return. The search returns the higher-priority (lower-numbered) entry.

For more information about target function libraries in the build process, refer to “Introduction to Target Function Libraries” in the Real-Time Workshop Embedded Coder documentation.

### Using a Processor-Specific Target Function Library to Optimize Code

As a best practice, you should select the appropriate target function library for your processor after you verify the ANSI C implementation of your project.

---

**Note** Do not select the processor-specific target function library if you use your executable application on more than one specific processor. The operator and function entries in a library may work on more than one processor within a processor family. The entries in a library usually do not work with different processor families.

---

To use target function library for processor-specific optimization when you generate code, you must install Real-Time Workshop Embedded Coder software. Your model must include a Target Preferences block configured for you intended processor.

Perform the following steps to select the target function library for your processor:

- 1 Select **Simulation > Configuration Parameters** from the model menu bar. The Configuration Parameters dialog box for your model opens.
- 2 On the **Select** tree in the Configuration Parameters dialog box, choose **Real-Time Workshop**.
- 3 Use **Browse** to select `idmlink_ert.tlc` as the **System target file**.
- 4 On the **Select** tree, choose **Interface**.

- 5** On the **Target function library** list, select the processor family that matches your processor. Then, click **OK** to save your changes and close the dialog box.

With the target function library selected, your generated code uses the specific functions in the library for your processor.

To stop using a processor-specific target function library, open the **Interface** pane in the model configuration parameters. Then, select the **C89/C90 (ANSI)** library from the **Target function library** list.

## **Process of Determining Optimization Effects Using Real-Time Profiling Capability**

You can use the real-time profiling capability to examine the results of applying the processor-specific library functions and operators to your generated code. After you select a processor-specific target function library, use the real-time execution profiling capability to examine the change in program execution time.

Use the following process to evaluate the effects of applying a processor-specific target function library when you generate code:

- 1** Enable real-time profiling in your model. Refer to “Profiling Code Execution in Real-Time” on page 5-10.
- 2** Generate code for your project using the default target function library **C89/C90 ANSI**.
- 3** Profile the code, and save the report.
- 4** Rebuild your project using a processor-specific target function library instead of the **C89/C90 ANSI** library.
- 5** Profile the code, and save the second report.
- 6** Compare the profile report from running your application with the processor-specific library selected to the profile results with the **ANSI** library selected in the first report.

For a demonstration of verifying the code optimization, refer to the Embedded IDE Link demos. Review the demo Optimizing Embedded Code via Target Function Library.

### Reviewing Processor-Specific Target Function Library Changes in Generated Code

Use one of the following techniques or tools to see the target function library elements where they appear in the generated code:

- Review the Code Manually.
- Use Model-to-Code Tracing to navigate from blocks in your model to the code generated from the block.
- Use a File Differencing Scheme to compare projects that you generate before and after you select a processor-specific target function library.

#### Reviewing Code Manually

To see where the generated code uses target function library replacements, review the file *modelName.c*. Look for code similar to the following examples.

For example, with CCS:For example, with MULTI:For example, with VisualDSP++:

The functions shown are the multiply implementation functions registered in the target function library. In these examples, the function performs an optimized multiplication operation. Similar functions appear for add, and sub. For more information about the arguments in the function, refer to “Introduction to Target Function Libraries” in the online Help system.

#### Using Model-to-Code Tracing

You can use the model-to-code report options in the configuration parameters to trace the code generated from any block with target function library. After you create your model and select a target function library, follow these steps to use the report options to trace the generated code:

- 1 Open the model configuration parameters.

- 2 Select **Report** from the **Select** tree.
- 3 In the Report pane, select **Create code generation report** and **Model-to-code**, and then save your changes.
- 4 Press **Ctrl+B** to generate code from your model.

The Real-Time Workshop Report window opens on your desktop. For more information about the report, refer to the “Generating Reports for Code Reviews and Traceability Analysis” topic in the Real-Time Workshop Embedded Coder documentation.

- 5 Use model-to-code highlighting to trace the code generated for each block with target function library applied:
  - Right-click on a block in your model and select **Real-Time Workshop > Navigate to code** from the context menu.
  - Select **Navigate-to-code** to highlight the code generated from the block in the report window.

Inspect the code to see the target function operator in the generated code. For more information, refer to “Tracing Code Generated Using Your Target Function Library” in the Real-Time Workshop Embedded Coder documentation in the online Help system.

If a target function library replacement did not occur as you expected, use the techniques described in “Examining and Validating Function Replacement Tables” in the Real-Time Workshop Embedded Coder documentation to help you determine why the build process did not use the function or operator.

## Using a File Differencing Scheme

You can also review the target function library induced changes in your project by comparing projects that you generate both with and without the processor-specific target function library.

- 1 Generate your project with the default C89/C90 ANSI target function library. Use **Create Project**, **Archive Library**, or **Build** for the **Build action** in the Embedded IDE Link options.
- 2 Save the project to a new name—*newproject1*.

- 3 Go back to the configuration parameters for your model, and select a target function library appropriate for your processor.
- 4 Regenerate your project.
- 5 Save the project with a new name—*newproject2*
- 6 Compare the contents of the *modelName.c* files from *newproject1* and *newproject2*. The differences between the files show the target function library induced code changes.

### Reviewing Target Function Library Operators and Functions

Real-Time Workshop Embedded Coder software provides the Target Function Library viewer to enable you to review the arithmetic operators and functions registered in target function library tables.

To open the viewer, enter the following command at the MATLAB prompt.

```
RTW.viewTf1
```

For details about using the target function library viewer, refer to “Selecting and Viewing Target Function Libraries” in the online Help system.

### Creating Your Own Target Function Library

For details about creating your own library, refer to the following sections in your Real-Time Workshop Embedded Coder documentation:

- “Introduction to Target Function Libraries”
- “Creating Function Replacement Tables”
- “Examining and Validating Function Replacement Tables”

## Model Reference

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Real-Time Workshop documentation provides much more information about model reference.

### How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models — Blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Real-Time Workshop documentation in the online Help system.

### Model Reference in Simulation

When you simulate the top model, Real-Time Workshop software detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates

an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink software rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these settings through the **Model Reference** pane of the Configuration Parameters dialog box.

### Model Reference in Code Generation

Real-Time Workshop software requires executables to generate code from models. If you have not simulated your model at least once, Real-Time Workshop software creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, Real-Time Workshop software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

### Using Model Reference

With few limitations or restrictions, Embedded IDE Link provides full support for generating code from models that use model reference.

#### Build Action Setting

The most important requirement for using model reference with the TI's processors is that you must set the **Build action** (go to **Configuration Parameters > Embedded IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action

- 1 Open your model.
- 2 Select **Simulation > Configuration Parameters** from the model menus.



The Configuration Parameters dialog box opens.

- 3** From the **Select** tree, choose **Embedded IDE Link**.
- 4** In the right pane, under **Runtime**, select set **Archive\_library** from the **Build action** list.

If your top model uses a reference model that does not have the build action set to **Archive\_library**, the build process automatically changes the build action to **Archive\_library** and issues a warning about the change.

As a result of selecting the **Archive\_library** setting, other options are disabled:

- With Texas Instruments CCS IDE, **DSP/BIOS** is disabled for all referenced models. Only the top model supports **DSP/BIOS** operation.
- **Overrun notification**, **Export IDE link handle to the base workspace**, and **System stack size** are disabled for the referenced models.

### Target Preferences Blocks in Reference Models

Each referenced model and the top model must include a **Target Preferences** block for the correct processor. You must configure all the **Target Preferences** blocks for the same processor.

To obtain information about which compiler to use and which archiver to use to build the referenced models, the referenced models require **Target Preferences** blocks. Without them, the compile and archive processes does not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the **Target Preferences** block in the model provides.

### Other Block Limitations

Model reference with **Embedded IDE Link** does not allow you to use certain blocks or **S-functions** in reference models:

- No blocks from the C62x DSP Library (in c6000lib) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (in c6000lib) (because these are noninlined S-functions)
- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Target Support Package or Target Support Package block library

## Configuring processors to Use Model Reference

processors that you plan to use in Model Referencing must meet some general requirements.

- A model reference compatible processor must be derived from the ERT or GRT processors.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation processor.
- The External mode option is not supported in model reference Real-Time Workshop software processor builds. Embedded IDE Link does not support External mode. If you select this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in Supporting Shared Utility Directories in the Build Process in the Real-Time Workshop documentation.

To use an existing processor, or a new processor, with Model Reference, you set the `ModelReferenceCompliant` flag for the processor. For information on how to set this option, refer to `ModelReferenceCompliant` in the online Help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference processor, use the following command to set the `ModelReferenceCompliant` flag to On:

```
set_param(bdroot, 'ModelReferenceCompliant', 'on')
```

Models that you develop with versions 2.4 and later of Embedded IDE Link automatically include the model reference capability. You do not need to set the flag.



# Generating Makefiles

---

- “Using Makefiles to Generate and Build Software” on page 4-2
- “Making an XMakefile Configuration Operational” on page 4-6
- “Example: Creating an XMakefile Configuration for the Intel Compiler” on page 4-7
- “XMakefile User Configuration Dialog Box” on page 4-17

## Using Makefiles to Generate and Build Software

In this section...
“Overview” on page 4-2
“Configuring Your Model to Use Makefiles” on page 4-2
“Choosing an XMakefile Configuration” on page 4-3
“Building Your Model” on page 4-5

### Overview

You can configure Embedded IDE Link to generate and build your software using makefiles. Scenarios for using this feature include:

- Building software without opening an IDE
- Automating the build process for testing and continuous build environments
- Fine-tuning and customizing the build process

### Configuring Your Model to Use Makefiles

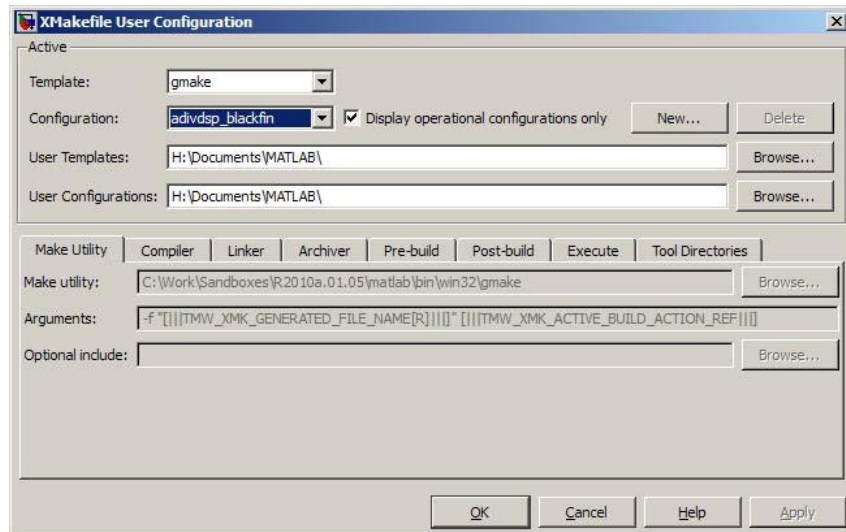
Update your model configuration parameters to use a makefile instead of an IDE when you build software from the model:

- 1** Add a target preferences block to your model and configure it for your target processor. For more information, see “Setting Target Preferences” on page 2-2.
- 2** In your model window, select **Simulation > Configuration Parameters**.
- 3** Under **Real-Time Workshop**, select **Embedded IDE Link**.
- 4** Set **Build format** to **Makefile**. For more information, see “Build format” on page 10-5.
- 5** Set **Build action** to **Build\_and\_execute**. For more information, see “Build action” on page 10-7.

## Choosing an XMakefile Configuration

Configure how Embedded IDE Link generates makefiles:

- 1 Enter `xmakefilessetup` on the MATLAB command line. The software opens an XMakefile User Configuration dialog box.



- 2 For **Template**, select the option that matches your make utility.
- 3 For **Configuration**, select the option that describes your software build toolchain and target processor platform.

---

**Note** If you set **Configuration** to `msvs_host`, restart MATLAB as described in “Working with Microsoft® Visual Studio” on page 4-4 before building your model software.

---

Things to consider while setting **Configuration**:

- Select **Display operational configurations only** to hide non-working configurations.

- To display all of the configurations, including non-operational configurations, deselect **Display operational configurations only**. For more information, see “Making an XMakefile Configuration Operational” on page 4-6.
- The list of configurations can include non-editable configurations defined in the software and editable configurations defined by you.
- To create a new editable configuration, use the **New** button.
- For more information, see “XMakefile User Configuration Dialog Box” on page 4-17.

### Working with Microsoft Visual Studio

If you set **Configuration** to `msvs_host`, restart MATLAB from a Visual Studio® command prompt before building your model software with makefiles. The `vsvars32.bat` file associated with the Visual Studio command prompt configures the Visual Studio environment. Starting MATLAB from this command prompt results in a session that can generate makefiles from the `msvs_host` configuration.

To restart MATLAB from a Visual Studio command prompt:

- 1 Open a Visual Studio command prompt:
  - a Select your MSVS product from the Windows® **Start > Programs** menu.
  - b In **Visual Studio Tools**, select the **Visual Studio Command Prompt**.  
For example:



- 2 Enter `matlab` at the **Visual Studio Command Prompt**.
- 3 In MATLAB, open and build your model.

If you do not restart MATLAB from Microsoft® Visual Studio command prompt, building your model software generates an error whose ending is similar to the following text:

```
The build failed with the following message:  
"C:/Program Files/Microsoft Visual Studio...
```



```
3792 Abort C:/Program Files/Microsoft Visual Studio 8/VC/bin/cl
gmake: *** [MW_cs1.obj] Error 134
```

A related article is available on the Microsoft Web site at:  
<http://msdn.microsoft.com/en-us/library/1700bbwd.aspx>

## Building Your Model

In your model, click the build button or enter **Ctrl+B**. Embedded IDE Link creates a makefile and performs the other actions you specified in **Build action**.

By default, Embedded IDE Link outputs the derived files in the <builddir>/<buildconfiguration> directory. For example, in model\_name/CustomMW.

---

**Note** With Green Hills MULTI, Embedded IDE Link outputs the derived files in the <builddir> directory. For example, in model\_ghsmulti.

---

# Making an XMakefile Configuration Operational

When the XMakefile utility starts, it checks each factory default configuration to verify that the specified toolchain paths exist. If the paths are invalid, the configuration is non-operational. Typically, the cause of this problem is a difference between the path in the configuration and the actual path of the IDE toolchain.

To make a configuration operational:

- 1** Deselect **Display operational configurations only** to display non-operational configurations.
- 2** Select the non-operational configuration from the **Configuration** options.
- 3** When you click **Apply**, a new dialog box prompts you for the directory path of any missing resources the configuration requires.

## Example: Creating an XMakefile Configuration for the Intel Compiler

### In this section...

“Overview” on page 4-7

“Create a Configuration” on page 4-7

“Modify the Configuration” on page 4-9

“Test the Configuration” on page 4-12

### Overview

This example shows you how to add makefile support for a software development toolchain to Embedded IDE Link. This example uses the Eclipse IDE, which provides an open framework and allows for otherwise unsupported toolchains.

### Create a Configuration

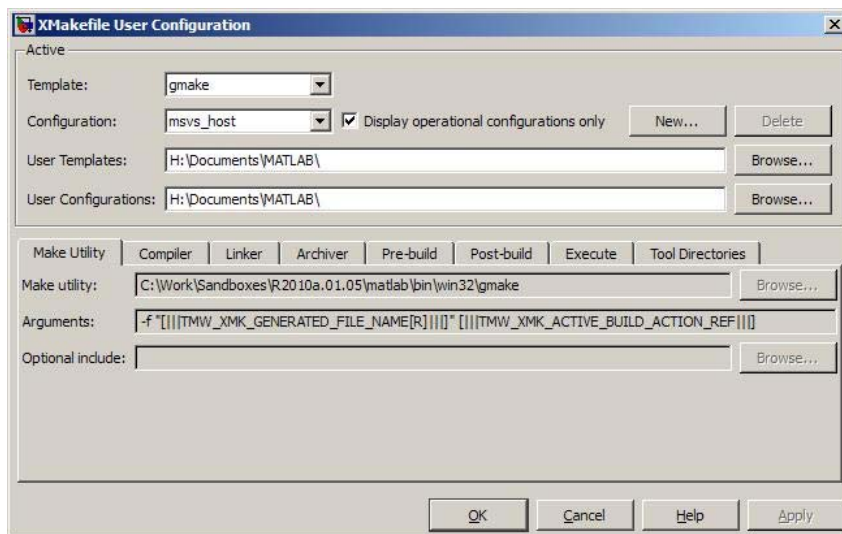
When you click **New**, the new configuration inherits values and behavior from the current configuration. Because of this inherited behavior and the fact that the Intel Compiler is not supported by any of the adaptors available in the Embedded IDE product; support for it can be added by cloning from any of these configurations: `msvs_host`, `mingw_host`, `montavista_arm` and `gcc_host`. These configurations provide an open framework and do not contain behaviors from the supported adaptors.

---

**Note** The linker used by the Intel Compiler uses the Microsoft Visual Studio tool chain and therefore the execution environment must have access to these tools (`vcvars.bat`). For more information, see “Working with Microsoft® Visual Studio” on page 4-4.

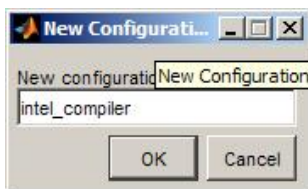
---

Open the XMakefile User Configuration UI by typing `xmakefilesetup` at the MATLAB prompt. This action displays the following dialog box.

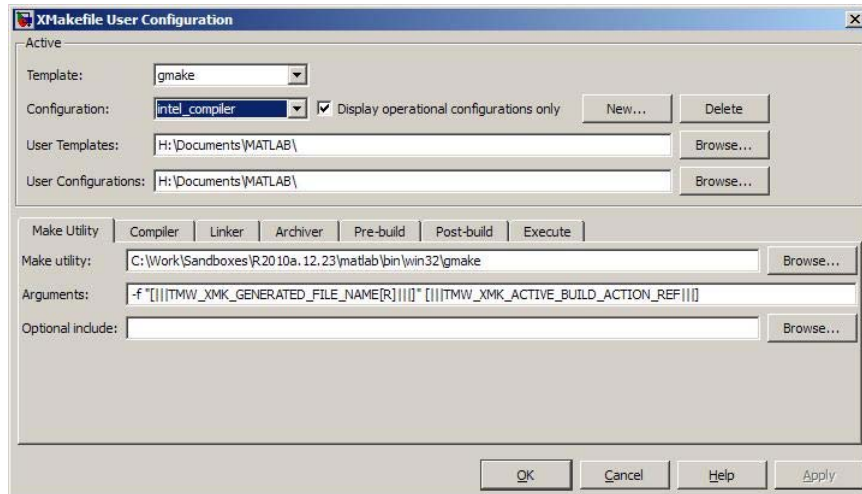


Select an existing configuration, such as `msvs_host`, `mingw_host`, `montavista_arm` or `gcc_host`. Click the **New** button.

A pop-up dialog prompts you for the name of the new configuration. Enter `intel_compiler` and click **OK**.



The dialog box displays a new configuration called `intel_compiler` based on `msvs_host`.

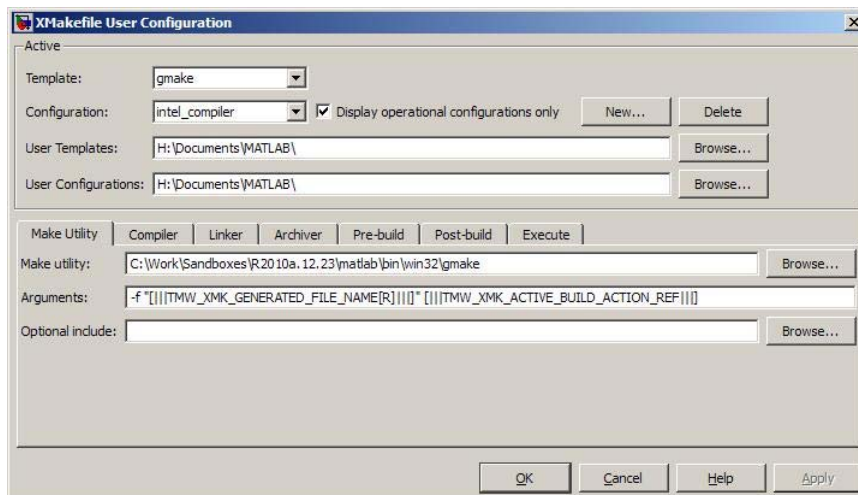


## Modify the Configuration

Adjust the settings of the newly created configuration. This example assumes the location of the Intel compiler is C:\Program Files\Intel\Compiler\.

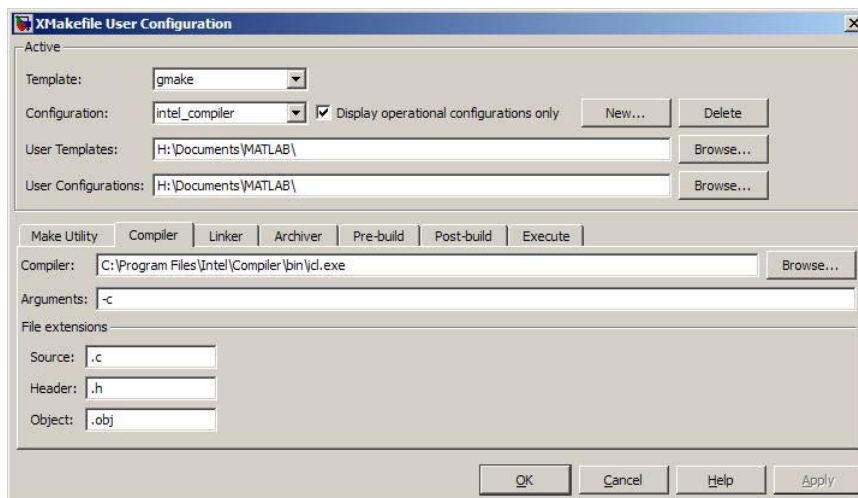
## Make Utility

You do not need to make any changes. This configuration uses the gmake tool that ships with MATLAB.



### Compiler

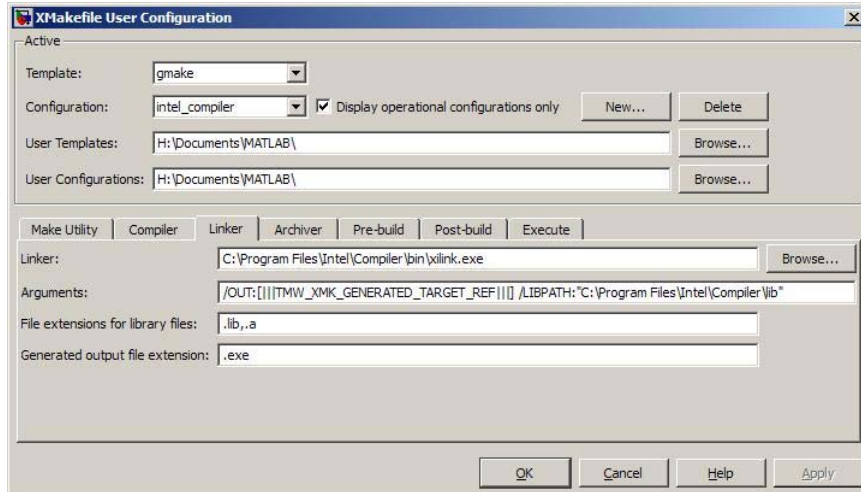
For **Compiler**, enter the location of `icl.exe` in the Intel installation.



### Linker

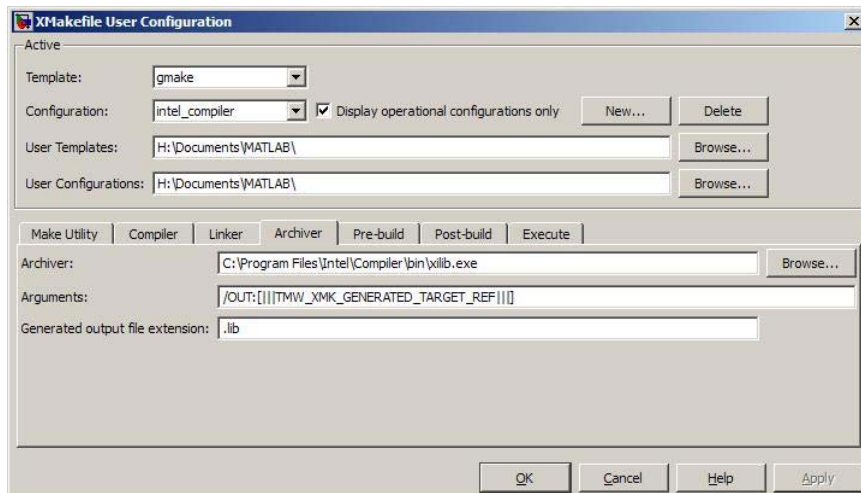
For **Linker**, enter the location of the linker executable, `xilink.exe`.

For **Arguments**, add the /LIBPATH path to the Intel libraries as shown here:



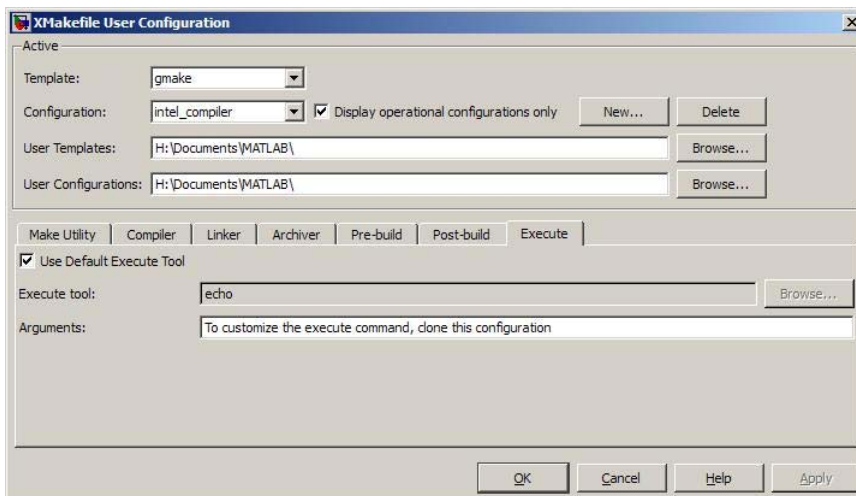
## Archiver

For **Archiver**, enter the location of the archiver, `xilib.exe`. Confirm that **File extensions for library files** includes `.lib`.



### Other tabs

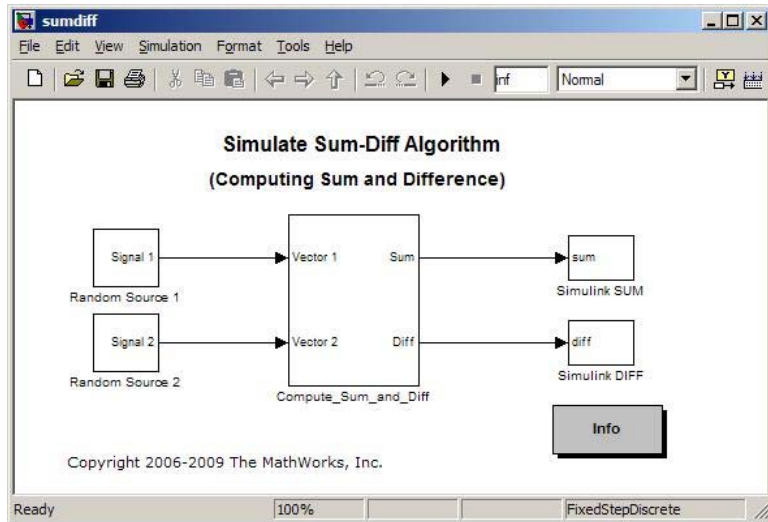
For this example, ignore the remaining tabs. In other circumstances, you can use them to configure additional build actions. For this example, the generated executable will be used as the “execute” target and later on the model used for testing will be configured to “Build\_and\_execute” on build action (CTRL-B).



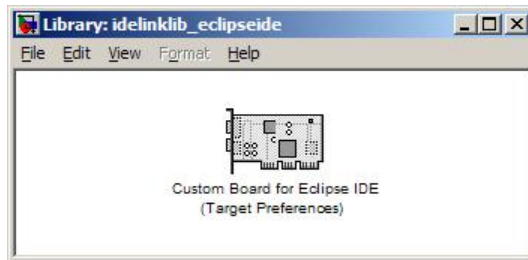
### Test the Configuration

Open the “sumdiff” model by entering `sumdiff` on the MATLAB prompt.

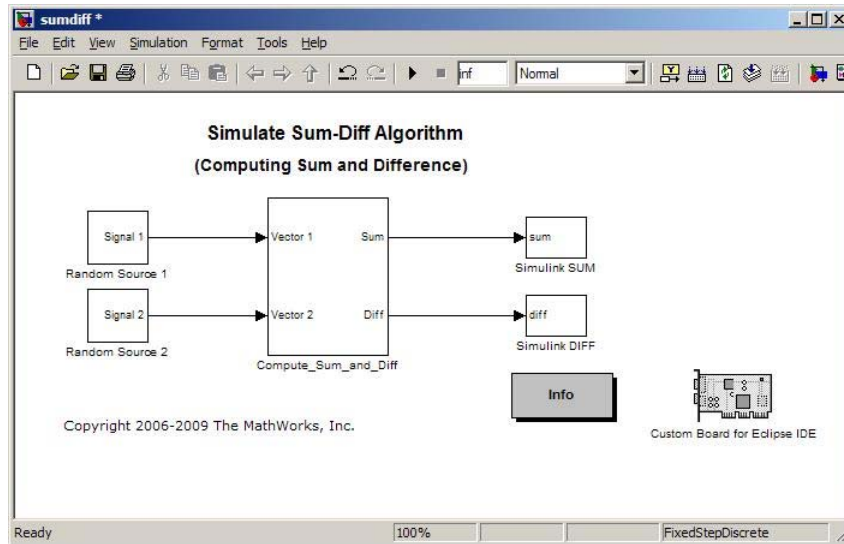




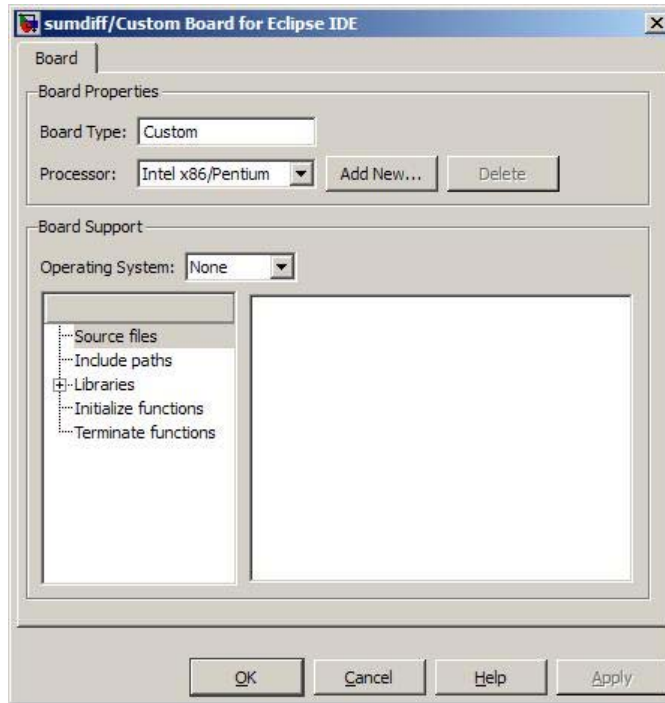
Configure the model for use with the Eclipse IDE. First open the Eclipse library by entering `idelinklib_eclipseide` at the MATLAB prompt.



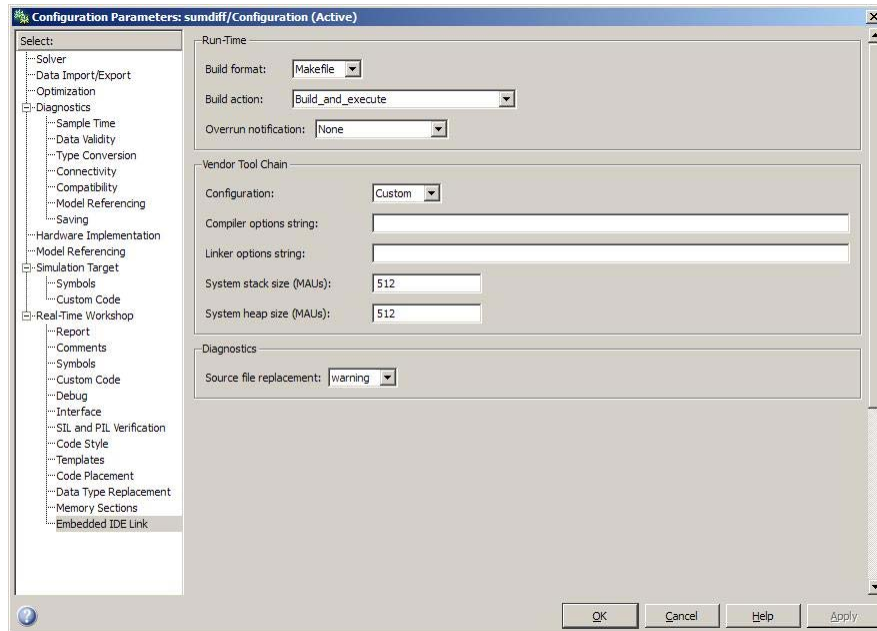
Then drag and drop the “Target preferences” block onto the `sumdiff` model.



Open the **Custom Board for Eclipse IDE** target preferences block. Set **Processor** to Intel x86/Pentium. Set **Operating System** to None or, if you have the Target Support Package product, select Windows. Click **OK**.



Open the Configuration Parameters for the sumdiff model by pressing **Ctrl+E**. Set **Build format** to **Makefile** and **Build action** to **Build\_and\_execute**.



Save the model to a temporary location, such as `C:\Temp\IntelTest\`.

Set that location as a working folder by typing `cd C:\temp\IntelTest\` at the MATLAB prompt.

Build the model by pressing **Ctrl+B**. The MATLAB Command Window displays something like:

```
### TLC code generation complete.
### Creating HTML report file sumdiff_codegen_rpt.html
### Creating project: c:\temp\IntelTest\sumdiff_eclipseide\sumdiff.mk
### Project creation done.
### Building project...
### Build done.
### Downloading program: c:\temp\IntelTest\sumdiff_eclipseide\sumdiff
### Download done.
```

A command window comes up showing the running model. Terminate the generated executable by pressing **Ctrl+C**.

## XMakefile User Configuration Dialog Box

### In this section...

“Active” on page 4-17

“Make Utility” on page 4-19

“Compiler” on page 4-20

“Linker” on page 4-21

“Archiver” on page 4-21

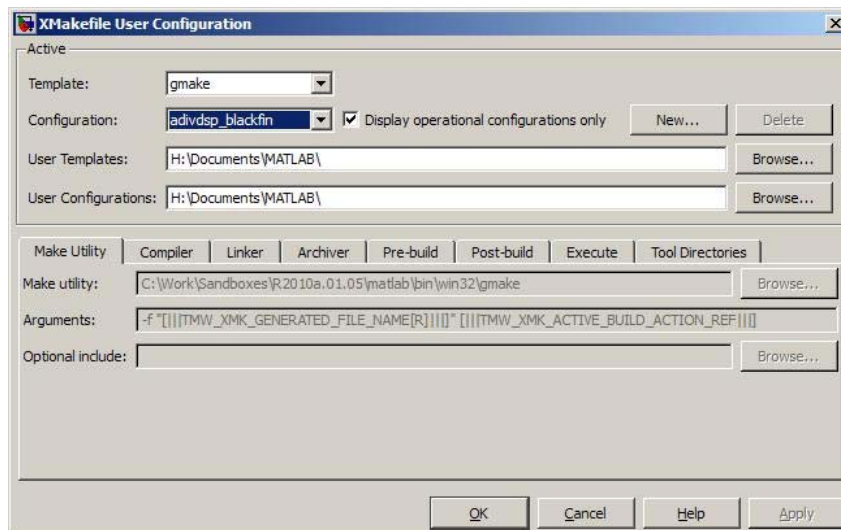
“Pre-build” on page 4-22

“Post-build” on page 4-23

“Execute” on page 4-23

“Tool Directories” on page 4-24

## Active



### Template

Select the template that matches your toolchain and processor. The template defines the syntax rules for writing the contents of the makefile or buildfile. The factory default template is gmake, which works with the GNU® make utility.

To add templates to this parameter, save them as .mkt files to the location specified by the **User Templates** parameter. For more information, see “User Templates” on page 4-19.

### Configuration

Select the configuration that best describes your toolchain and embedded processor platform.

You cannot edit or delete the factory default configurations provided by The MathWorks™. You can, however, edit and delete the configurations that you create.

Use the **New** button to create an editable copy of the currently selected configuration.

Use the **Delete** button to delete a configuration you created.

---

**Note** You cannot edit or delete the factory default configurations provided by The MathWorks.

---

### Display operational configurations only

When you open the XMakefile User Configuration dialog box, the software verifies that each factory default configuration contains valid paths to the executable files it uses.

To display valid factory default configurations, select “Display operational configurations only”. This option does not apply to configurations you create.

To display all of the configurations, including non-operational configurations, deselect **Display operational configurations only**. The software

categorizes a configuration as non-operational if a required resource is missing. For more information, see “Making an XMakefile Configuration Operational” on page 4-6.

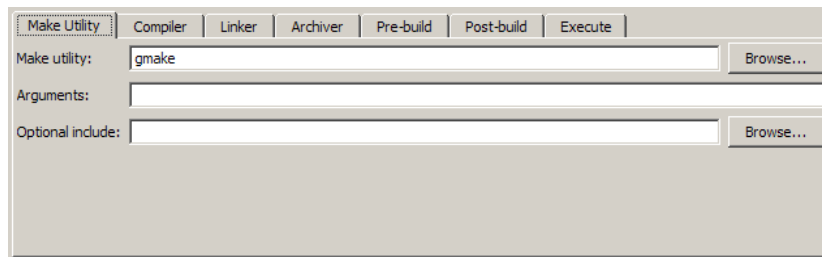
## User Templates

Set the path of the folder to which you can add template files. Saving templates files with the .mkt extension to this folder adds them to the **Templates** options.

## User Configurations

Set the location of configuration files you create with the **New** button.

## Make Utility



The screenshot shows the 'Make Utility' tab of the XMakefile User Configuration Dialog Box. The dialog has a title bar with tabs for 'Make Utility', 'Compiler', 'Linker', 'Archiver', 'Pre-build', 'Post-build', and 'Execute'. The 'Make Utility' tab is active. It contains three input fields: 'Make utility:' with the text 'gmake' and a 'Browse...' button; 'Arguments:' with an empty text box; and 'Optional include:' with an empty text box and a 'Browse...' button. The background of the dialog is a light gray color.

## Make utility

Set the path and filename of the make utility executable.

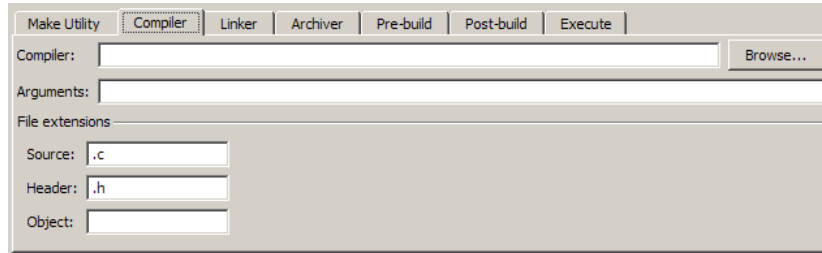
## Arguments

Define the command-line arguments to pass to the make utility. For more information, consult the third-party documentation for your make utility.

## Optional include

Set the path and file name of an optional include file.

### Compiler



The screenshot shows a dialog box titled 'Make Utility' with several tabs: 'Make Utility', 'Compiler', 'Linker', 'Archiver', 'Pre-build', 'Post-build', and 'Execute'. The 'Compiler' tab is active. It contains the following fields:

- Compiler:** A text input field with a 'Browse...' button to its right.
- Arguments:** A text input field.
- File extensions:** A section containing three text input fields:
  - Source:** Contains the text '.c'.
  - Header:** Contains the text '.h'.
  - Object:** An empty text input field.

### Compiler

Set the path and file name of the compiler executable.

### Arguments

Define the command-line arguments to pass to the compiler. For more information, consult the third-party documentation for your compiler.

### Source

Define the file name extension for the source files. Use commas to separate multiple file extensions.

### Header

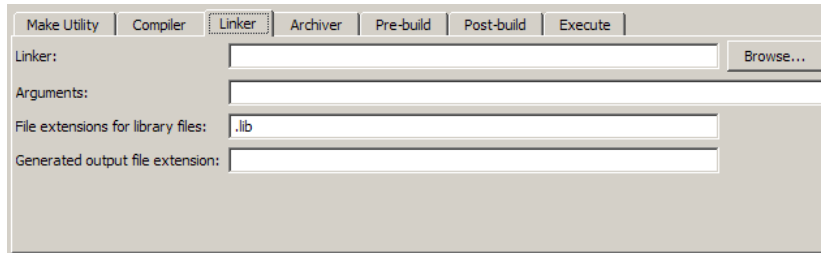
Define the file name extension for the header files. Use commas to separate multiple file extensions.

### Object

Define the file name extension for the object files.



## Linker



The screenshot shows the 'Linker' tab of the XMakefile User Configuration Dialog Box. The dialog has a title bar with tabs for 'Make Utility', 'Compiler', 'Linker', 'Archiver', 'Pre-build', 'Post-build', and 'Execute'. The 'Linker' tab is active. It contains four input fields: 'Linker:' with a 'Browse...' button, 'Arguments:', 'File extensions for library files:' containing '.lib', and 'Generated output file extension:'.

### Linker

Set the path and file name of the linker executable.

### Arguments

Define the command-line arguments to pass to the linker. For more information, consult the third-party documentation for your linker.

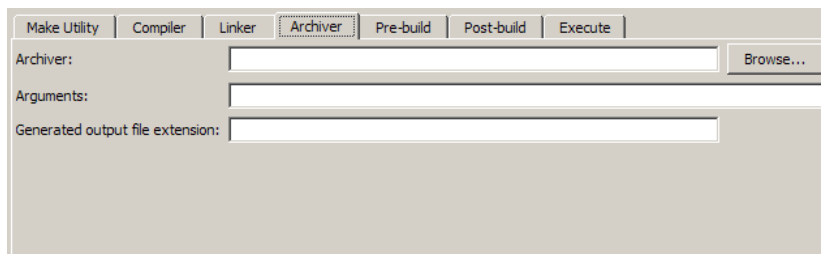
### File extensions for library files

Define the file name extension for the file library files. Use commas to separate multiple file extensions.

### Generated output file extension

Define the file name extension for the generated libraries or executables.

## Archiver



The screenshot shows the 'Archiver' tab of the XMakefile User Configuration Dialog Box. The dialog has a title bar with tabs for 'Make Utility', 'Compiler', 'Linker', 'Archiver', 'Pre-build', 'Post-build', and 'Execute'. The 'Archiver' tab is active. It contains three input fields: 'Archiver:' with a 'Browse...' button, 'Arguments:', and 'Generated output file extension:'.

### Archiver

Set the path and file name of the archiver executable.

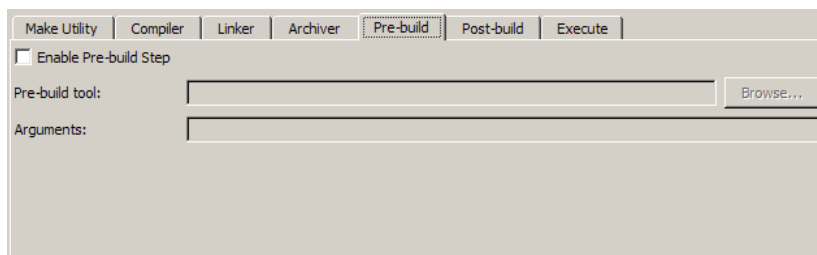
### Arguments

Define the command-line arguments to pass to the archiver. For more information, consult the third-party documentation for your archiver.

### Generated output file extension

Define the file name extension for the generated libraries.

## Pre-build



### Enable Prebuild Step

Select this check box to define a prebuild tool that runs before the compiler.

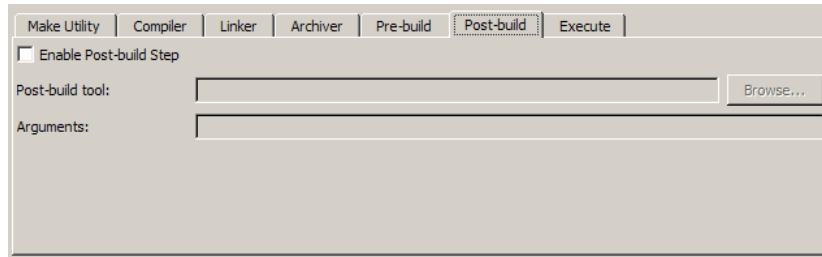
### Prebuild tool

Set the path and file name of the prebuild tool executable.

### Arguments

Define the command-line arguments to pass to the prebuild tool. For more information, consult the third-party documentation for your prebuild tool.

## Post-build



The screenshot shows the 'Post-build' tab of the XMakefile User Configuration Dialog Box. The tab is selected and highlighted. The dialog box contains a check box labeled 'Enable Post-build Step'. Below this, there are two text input fields: 'Post-build tool:' and 'Arguments:'. The 'Post-build tool:' field has a 'Browse...' button to its right. The 'Arguments:' field is a larger text area.

### Enable Postbuild Step

Select this check box to define a postbuild tool that runs after the compiler or linker.

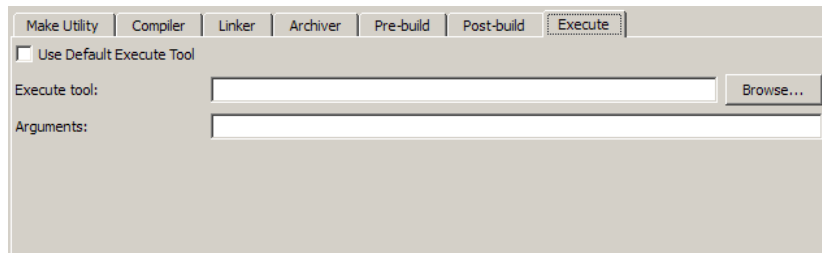
### Postbuild tool

Set the path and file name of the postbuild tool executable.

### Arguments

Define the command-line arguments to pass to the postbuild tool. For more information, consult the third-party documentation for your postbuild tool.

## Execute



The screenshot shows the 'Execute' tab of the XMakefile User Configuration Dialog Box. The tab is selected and highlighted. The dialog box contains a check box labeled 'Use Default Execute Tool'. Below this, there are two text input fields: 'Execute tool:' and 'Arguments:'. The 'Execute tool:' field has a 'Browse...' button to its right. The 'Arguments:' field is a larger text area.

### Use Default Execute Tool

Select this check box to use the generated derivative as the execute tool when the build process is complete. Uncheck it to specify a different tool.

The default value, `echo`, simply displays a message that the build process is complete.

---

**Note** On Linux®, multirate multitasking executables require root privileges to schedule POSIX threads with real-time priority. If you are using makefiles to build multirate multitasking executables on your Linux development system, you cannot use **Execute tool** to run the executable. Instead, use the Linux command, `sudo`, to run the executable.

---

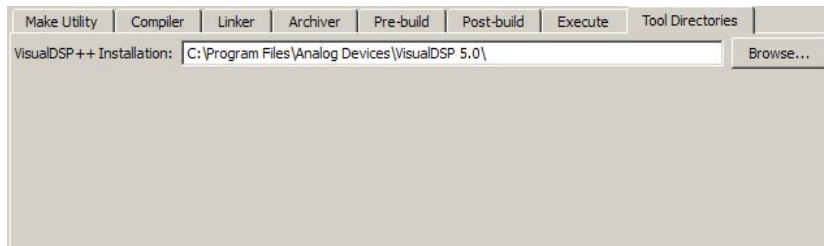
### Execute tool

Set the path and file name of the execute tool executable or built-in command.

### Arguments

Define the command-line arguments to pass to the execute tool. For more information, consult the third-party documentation for your execute tool.

### Tool Directories



### Installation

Use the Tool Directories tab to change the toolchain path of an operational configuration.

For example, if you installed two versions of an IDE in separate directories, you can use the **Installation** path to change which IDE-related build tools the configuration uses.

# Verifying Generated Code

---

- “What Is Verification?” on page 5-2
- “Verifying Generated Code via Processor-in-the-Loop” on page 5-3
- “Profiling Code Execution in Real-Time” on page 5-10
- “System Stack Profiling” on page 5-18

### What Is Verification?

Verification consists broadly of running generated code on a processor and verifying that the code does what you intend. The components of Embedded IDE Link combine to provide tools that help you verify your code during development by letting you run portions of simulations on your hardware and profiling the executing code.

Using the Automation Interface and Project Generator components, Embedded IDE Link offers the following verification functions:

- Processor-in-the-Loop — A technique to help you evaluate how your process runs on your processor
- Real-Time Task Execution Profiling — A tool that lets you see how the tasks in your process run in real-time on your processor hardware

## Verifying Generated Code via Processor-in-the-Loop

### In this section...

“What is Processor-in-the-Loop?” on page 5-3

“Using the Top-Model PIL Approach” on page 5-5

“Using the PIL Block Approach” on page 5-6

“Definitions” on page 5-8

“Other Aspects of PIL” on page 5-9

“PIL Issues and Limitations” on page 5-9

### What is Processor-in-the-Loop?

PIL is a feature of the Real-Time Workshop Embedded Coder product. To use PIL in Embedded IDE Link, you must have a Real-Time Workshop Embedded Coder license.

You can use processor-in-the-loop (PIL) simulation to verify your generated code on a target processor or instruction set simulator. In PIL simulation, the target processor participates fully in the simulation loop — hence the term processor-in-the-loop simulation. You can compare the output of regular simulation modes, such as Normal or Accelerator, and PIL simulation mode to verify your generated code. You can easily switch between simulation and PIL modes. This flexibility allows you to verify the generated code by executing the model as compiled code in the target environment. You can model and test your embedded software component in Simulink and then reuse your regression test suites across simulation and compiled object code. This avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for the production hardware.

For complete information about PIL, see “Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder User’s Guide. For an example of how to use PIL, see the PIL: Fault-Tolerant Fuel Control System demo for your IDE.

Embedded IDE Link supports two PIL approaches:

- Top-model PIL
- PIL block

Embedded IDE Link does not support the “Model Block PIL” approach.

---

**Note** Processor-in-the-Loop (PIL) builds and uses a MEX function to run the PIL cosimulation block. Before using PIL, set up a compiler for MATLAB to build the MEX files. Run the command `|mex -setup|` to select a compiler configuration. For more information, read “Building MEX-Files”

---

### **When to Use Top-Model PIL**

Use top-model PIL if you want to:

- Verify code generated for a top model (standalone code interface).
- Load test vectors or stimulus inputs from the MATLAB workspace.
- Switch the entire model between normal and SIL or PIL simulation modes.

For more information, see “Choosing a PIL Approach”

### **When to Use the PIL Block**

Use the PIL Block if you want to:

- Use a compiler and target environment supported by the Embedded IDE Link product.
- Verify code generated for a top model (standalone code interface) or subsystem (right-click build standalone code interface).
- Change the model and insert a PIL block to represent a component running in SIL or PIL mode, and the test harness model or a system model provides test vector or stimulus inputs.

For more information, see “Choosing a PIL Approach”.



## Using the Top-Model PIL Approach

### Setting Model Configuration Parameters to Generate the PIL Application

Configure your model to generate the PIL executable from your model:

- 1 Add a Target Preferences block to your model from one of the following libraries: , , , and .
- 2 Open the Target Preferences block and select your processor from the list of processors. For more information, refer to Target Preferences/Custom Board
- 3 From the model window, select **Simulation > Configuration Parameters**.
- 4 In Configuration Parameters, select **Real-Time Workshop**.
- 5 Set **System Target File** to `idmlink_ert.tlc`.
- 6 On the **Select** tree, choose **Embedded IDE Link**.
- 7 Set **Build format** to Project.
- 8 Set **Build action** to `Create_processor_in_the_loop_project`.
- 9 Click **OK** to close the Configuration Parameters dialog box.

### Running the Top Model PIL Application

To create a PIL block, perform the following steps:

- 1 In the model window menu, select **Simulation > Processor-in-the-loop**.
- 2 In the model toolbar, click the Start simulation button.

A new model window opens and the new PIL model block appears in it. The third-party IDE compiles and links the PIL executable file. Follow the progress of the build process in the MATLAB command window.

## Using the PIL Block Approach

### Preparing Your Model to Generate a PIL Block

Start with a model that contains the algorithm blocks you want to verify on the processor as compiled object code. To create a PIL application and PIL block from your algorithm subsystem, follow these steps:

- 1 Identify the algorithm blocks to co-simulate.
- 2 Convert those blocks into an unmasked subsystem in your model.

For information about how to convert your process to a subsystem, refer to Creating Subsystems in *Using Simulink* or in the online Help system.

- 3 Open the newly created subsystem and copy a Target Preferences block to it from one of the following libraries: , , , and .

Open the Target Preferences block and select your processor from the list of processors. For more information, refer to Target Preferences/Custom Board

### Setting Model Configuration Parameters to Generate the PIL Application

After you create your subsystem, set the configuration parameters for your model to enable the model to generate a PIL block.

Configure your model to enable it to generate PIL algorithm code and a PIL block from your subsystem:

- 1 From the model menu bar, select **Simulation > Configuration Parameters**. This opens Configuration Parameters dialog box.
- 2 On the **Select** tree, choose **Real-Time Workshop**.
- 3 Set **System Target File** to `idmlink_ert.tlc`.
- 4 On the **Select** tree, choose **Embedded IDE Link**.
- 5 Set **Build format** to Project.

- 6** Set **Build action** to `Create_processor_in_the_loop_project`. The **PIL block action** option appears.
- 7** Set **PIL block action** to `Create_PIL_block_build_and_download`.
- 8** Click **OK** to close the Configuration Parameters dialog box.

## Creating the PIL Block Application from a Model Subsystem

To create a PIL block, perform the following steps:

- 1** Right-click the masked subsystem in your model and select **Real-Time Workshop > Build Subsystem** from the context menu.

A new model window opens and the new PIL block appears in it. The third-party IDE compiles and links the PIL executable file.

This step builds the PIL algorithm object code and a PIL block that corresponds to the subsystem, with the same inputs and outputs. Follow the progress of the build process in the MATLAB command window.

- 2** Copy the new PIL block from the new model to your model. To simulate the subsystem processes concurrently, place it parallel to your masked subsystem. Otherwise, replace the subsystem with the PIL block.

To see a PIL block in a parallel masked subsystem, see the *Getting Started with Application Development* demo for your IDE among the demos.

---

**Note** Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and incorrect results.

---

## Running Your PIL Application to Perform Cosimulation and Verification

After you add your PIL block to your model, click **Simulation > Start** to run the PIL simulation and view the results.

## Definitions

### Cosimulation

The division of model simulation activities between a workstation and embedded processor. The test harness runs on the workstation and drives the inputs to the algorithm. The code generated from the algorithm runs on the embedded processor.

### PIL Algorithm

The algorithmic code, which corresponds to a subsystem or portion of a model, to test during the PIL cosimulation. The PIL algorithm is in compiled object form to enable verification at the object level.

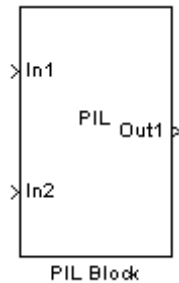
### PIL Application

The executable application that runs on the processor platform. Embedded IDE Link creates a PIL application by augmenting your algorithmic code with the PIL execution framework. The PIL execution framework code is then compiled as part of your embedded application.

The PIL execution framework code includes the `string.h` header file so that the PIL application can use the `memcpy` function. The PIL application uses `memcpy` to exchange data between the Simulink model and the cosimulation processor.

### PIL Block

When you build a subsystem from a model for PIL, the process creates a PIL block optimized for PIL cosimulation. When you run the simulation, the PIL block acts as the interface between the model and the PIL application running on the processor. The PIL block inherits the shape and signal names from the source subsystem in your model, as shown in the following example. Inheritance is convenient for copying the PIL block into the model to replace the original subsystem for cosimulation.



## Other Aspects of PIL

### PIL Issues and Limitations

Consider the following issues when you work with PIL blocks.

#### Generic PIL Issues

Refer to the Support Table section in the Real-Time Workshop Embedded Coder documentation for general information about using the PIL block with embedded link products. Refer to PIL Feature Support and Limitations.

#### Real-Time Workshop grt.tlc-Based Targets Not Supported

PIL does not support grt.tlc system target files.

To use PIL, set **System target file** option to `idmlink_ert.tlc`. (The **System target file** option is located on the Configuration Parameters > Real-Time Workshop pane).

## Profiling Code Execution in Real-Time

In this section...
“Overview” on page 5-10
“Profiling Execution by Tasks” on page 5-11
“Profiling Execution by Subsystems” on page 5-13

### Overview

Real-time execution profiling in Embedded IDE Link software uses a set of utilities to support profiling for synchronous and asynchronous tasks, or atomic subsystems, in your generated code. These utilities record, upload, and analyze the execution profile data.

Execution profiler supports profiling your code two ways:

- Tasks—Profile your project according to the tasks in the code.
- Atomic subsystems—Profile your project according to the atomic subsystems in your model.

---

**Note** To perform execution profiling, you must generate your project from a model in Simulink modeling environment.

---

When you enable profiling, you select whether to profile by task or subsystem.

To profile by subsystems, you must configure your model with at least one atomic subsystem. To learn more about creating atomic subsystems, refer to “Creating Subsystems” in the online help for Simulink software.

The profiler generates output in the following formats:

- Graphical display that shows task or subsystem activation, preemption, resumption, and completion. All data appears in a MATLAB graphic with the data notated by model rates or subsystems and execution time.

- An HTML report that provides statistical data about the execution of each task or atomic subsystem in the running process.

These reports are identical to the reports you see if you use `profile(IDE_Obj, 'execution', 'report')` to view the execution results. For more information about report formats, refer to `profile`. In combination, the reports provide a detailed analysis of how your code runs on the processor.

Use this general process for profiling your project:

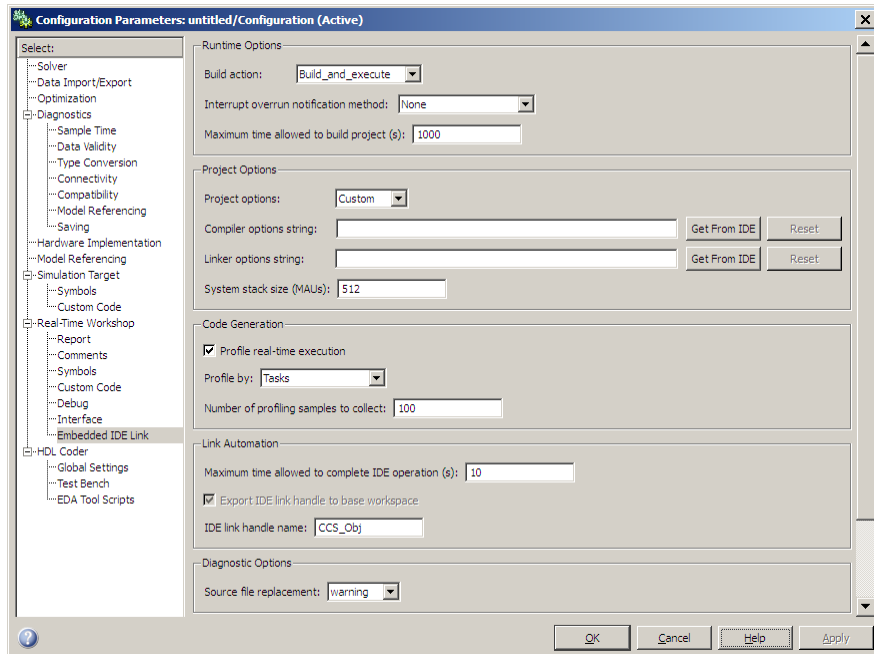
- 1** Create your model in Simulink modeling environment.
- 2** Enable execution profiling in the configuration parameters for your model.
- 3** Run your application.
- 4** Stop your application.
- 5** Get the profiling results with the `profile` function.

The following sections describe profiling your projects in more detail.

## Profiling Execution by Tasks

To configure a model to use task execution profiling, perform the following steps:


- 1** Open the Configuration Parameters dialog box for your model.
- 2** Select Embedded IDE Link from the **Select** tree.
- 3** Select **Profile real-time execution**.
- 4** On the **Profile by** list, select **Tasks** to enable real-time task profiling.



**5** By default, the **Export IDE link handle to base workspace** is enabled, and the **IDE link handle name** is set to `IDE_Obj`.

**6** Click **OK** to close the Configuration Parameters dialog box.

To view the execution profile for your model:

- 1** Click **Incremental build** () on the model toolbar to generate, build, load, and run your code on the processor.
- 2** To stop the running program, select **Debug > Halt** in the IDE or use `IDE_obj.halt` from the MATLAB command prompt. Gathering profiling data from a running program may yield incorrect results.
- 3** At the MATLAB command prompt, enter

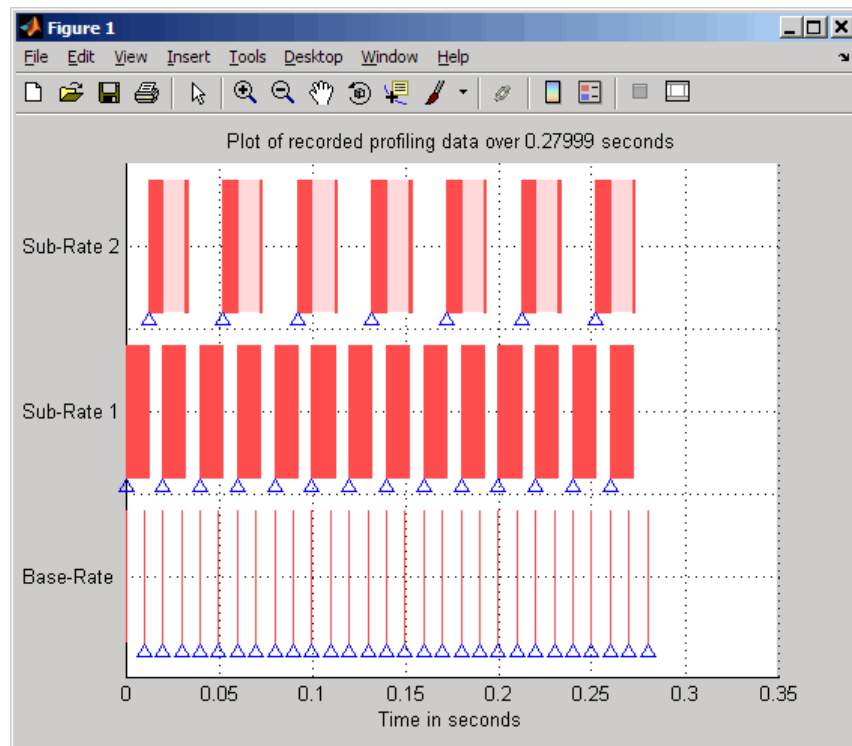
```
profile(handlename, execution , report )
```



to view the MATLAB software graphic of the execution report and the HTML execution report.

Refer to `profile` for information about other reporting options.

The following figure shows the profiling plot from running an application that has three rates—the base rate and two slower rates. The gaps in the Sub-Rate2 task bars indicate preempted operations.

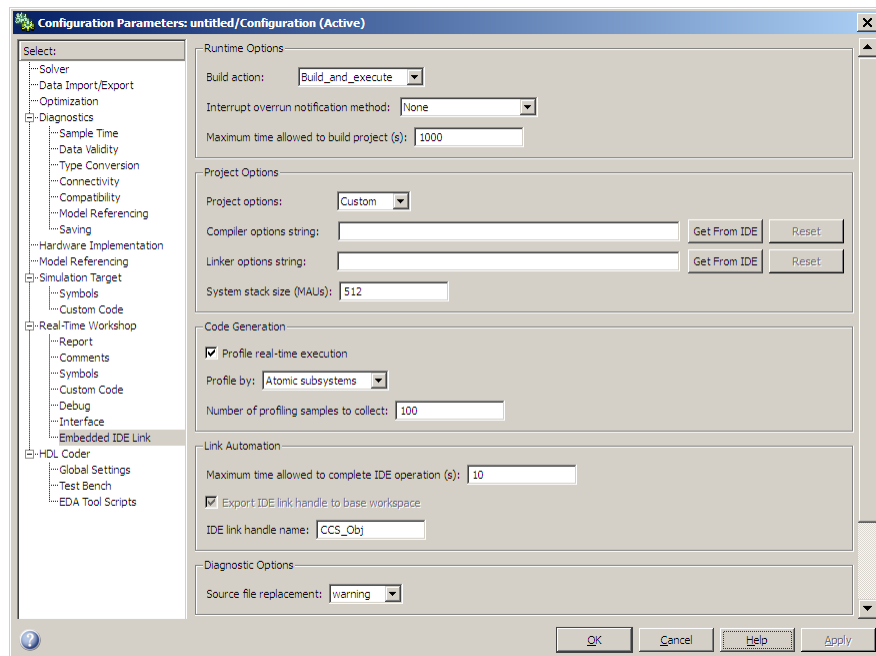


## Profiling Execution by Subsystems

When your models use atomic subsystems, you have the option of profiling your code based on the subsystems along with the tasks.


To configure a model to use subsystem execution profiling, perform the following steps:

- 1 Open the Configuration Parameters dialog box for your model.
- 2 Select **Embedded IDE Link** from the **Select** tree. The pane appears as shown in the following figure.
- 3 Select **Profile real-time execution**.
- 4 On the **Profile by** list, select **Atomic subsystems** to enable real-time subsystem execution profiling.



- 5 By default, the **Export IDE link handle to base workspace** is enabled, and the **IDE link handle name** is set to `IDE_Obj`.
- 6 Click **OK** to close the Configuration Parameters dialog box.

To view the execution profile for your model:

- 1 Click **Incremental build** () on the model toolbar to generate, build, load, and run your code on the processor.

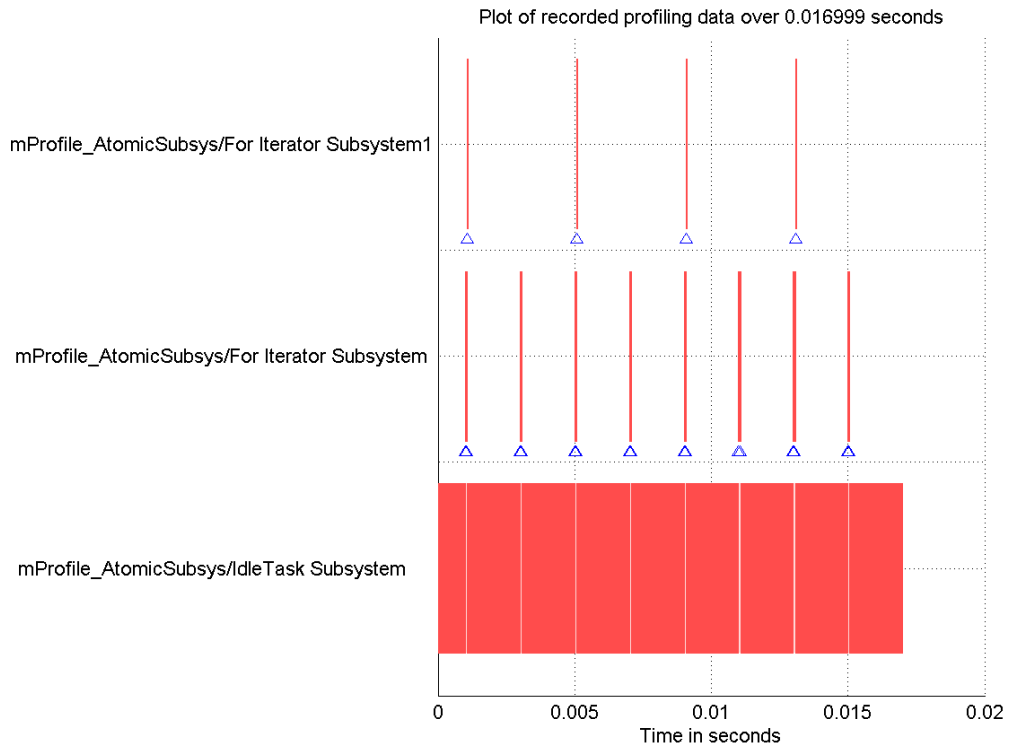
- 2** To stop the running program, select **Debug > Halt** in the IDE, or use `halt(handlename)` from the MATLAB command prompt. Gathering profile data from a running program may yield incorrect results.
- 3** At the MATLAB command prompt, enter:

```
profile(handlename, execution , report )
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

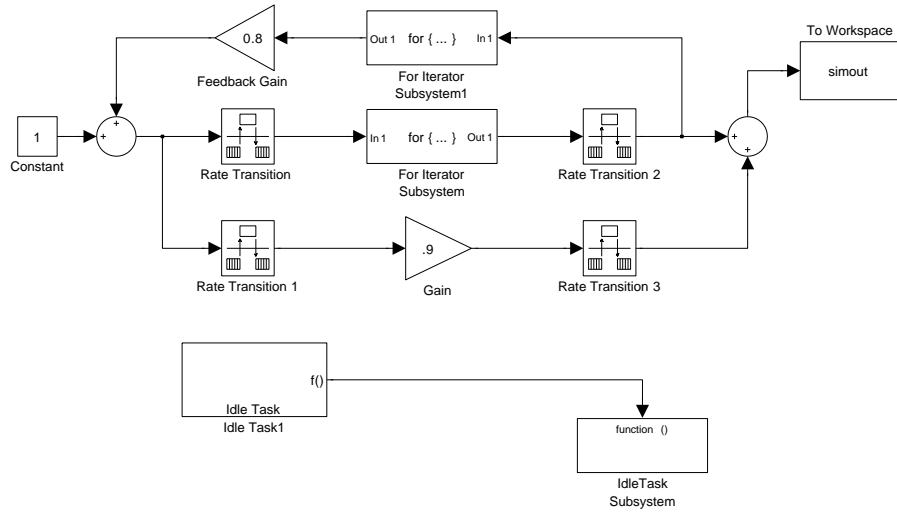
Refer to `profile` for more information.

The following figure shows the profiling plot from running an application that has three subsystems—For Iterator Subsystem, For Iterator Subsystem1, and Idle Task Subsystem.



The following figure presents the model that contains the subsystems reported in the profiling plot.

### Atomic Subsystem Profiling



## System Stack Profiling

In this section...
“Overview” on page 5-18
“Profiling System Stack Use” on page 5-20

### Overview

Embedded IDE Link software enables you to determine how your application uses the processor system stack. Using the `profile` method, you can initialize and test the size and usage of the stack. This information can help you optimize both the size of the stack and how your code uses the stack.

To provide stack profiling, `profile` writes a known pattern to the addresses in the stack. After you run your application for a while, and then stop your application, `profile` examines the contents of the stack addresses. `profile` counts each address that no longer contains the known pattern as used. The total number of address that have been used, compared to the total number of addresses you allocated, becomes the stack usage profile. This profile process does not tell you how often any address was changed by your application.

You can profile the stack with both the manually written code in a project and the code you generate from a model.

---

**Note** With Texas Instruments CCS IDE, if your project uses DSP/BIOS, stack profiling always reports 100% stack usage.

---

When you use `profile` to initialize and test the stack operation, the software returns a report that contains information about stack size, usage, addresses, and direction. With this information, you can modify your code to use the stack efficiently. The following program listing shows the stack usage results from running an application on a simulator.

```
profile(IDE_Obj, 'stack', 'report')
```

```
Maximum stack usage:
```

System Stack: 532/1024 (51.95%) MAUs used.

```

name: System Stack
startAddress: [512 0]
endAddress: [1535 0]
stackSize: 1024 MAUs
growthDirection: ascending

```

The following table describes the entries in the report:

Report Entry	Units	Description
System Stack	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
name	String for the stack name	Lists the name assigned to the stack.
startAddress	Decimal address and page	Lists the address of the stack start and the memory page.
endAddress	Decimal address and page	Lists the address of the end of the stack and the memory page.
stackSize	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
growthDirection	Not applicable	Reports whether the stack grows from the lower address to the higher address (ascending) or from higher to lower (descending).

### Profiling System Stack Use

To profile the system stack operation, perform these tasks in order:

- 1 Load an application.
- 2 Set up the stack to enable profiling.
- 3 Run your application.
- 4 Request the stack profile information.

---

**Note** With Texas Instruments CCS IDE, if your application initializes the stack with known values when you run it, stack usage is reported as 100%. The value does not correctly reflect the stack usage. For example, DSP/BIOS™ writes a fixed pattern to the stack (0x00C0FFEE) when you run your project. This pattern prevents the stack profiler from reporting the stack usage correctly. Disable DSP/BIOS to use stack profiling in your project development.

---

Follow these steps to profile the stack as your application interacts with it. This particular example uses a IDE handle object, `IDE_Obj`, for Texas Instruments' Code Composer Studio. However, you can generalize from this example to any IDE that supports profiling.

- 1 Load the application to profile.
- 2 Use the `profile` method with the **setup** input keyword to initialize the stack to a known state.

```
profile(IDE_Obj, 'stack', 'setup')
```

With the **setup** input argument, `profile` writes a known pattern into the addresses that compose the stack. For example, the pattern for C6000 processors is `A5`, and for C2000 and C5000 processors is `A5A5` (to account for their address size). As long as your target application does not write the same pattern to the system stack, `profile` can report the stack usage correctly.

- 3 Run your application.



- 4** Stop your running application. Stack use results gathered from an application that is running may be incorrect.
- 5** Use the `profile` method to capture and view the results of profiling the stack.

```
profile(IDE_Obj, 'stack', 'report')
```

The following example demonstrates setting up and profiling the stack. The IDE handle object, `IDE_Obj`, must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a TI C6713 simulator.

```
profile(IDE_Obj, 'stack', 'setup') % Set up processor stack--write A5 to the stack addresses.
```

```
Maximum stack usage:
```

```
System Stack: 0/1024 (0%) MAUs used.
```

```
      name: System Stack
startAddress: [512  0]
endAddress: [1535  0]
stackSize: 1024 MAUs
growthDirection: ascending
```

```
run(IDE_Obj)
```

```
halt(IDE_Obj)
```

```
profile(IDE_Obj, 'stack', 'report') % Request stack use report.
```

```
Maximum stack usage:
```

```
System Stack: 356/1024 (34.77%) MAUs used.
```

```
      name: System Stack
startAddress: [512  0]
endAddress: [1535  0]
stackSize: 1024 MAUs
growthDirection: ascending
```



# Block Reference

---

The blocks are grouped by library. To open the block library type the block library command, shown in parentheses, at the MATLAB command line.

Block Library: `idelinklib_common`  
(p. 6-2)

Blocks for use with ADI  
VisualDSP++, GHS MULTI,  
and TI Code Composer Studio

## **Block Library: idelinklib\_common**

Idle Task

Memory Allocate

Memory Copy

Create free-running task

Allocate memory section

Copy to and from memory section

# Blocks — Alphabetical List

---

# Idle Task

---

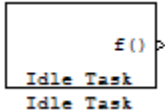
## Purpose

Create free-running task

## Library

Block Library: idelinklib\_common

## Description



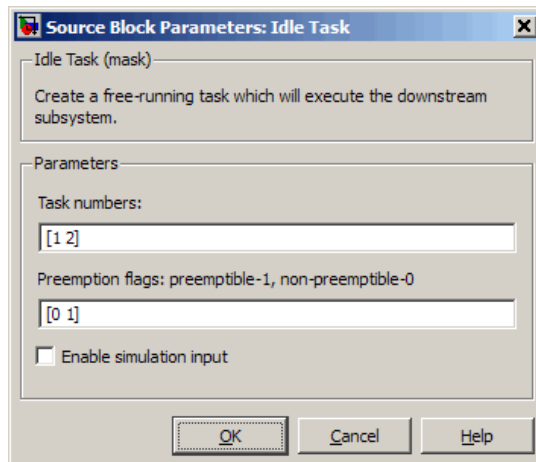
The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. All tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

## 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.

# Idle Task

---

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.

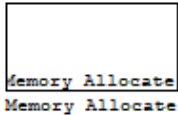
---



**Purpose** Allocate memory section

**Library** Block Library: idelinklib\_common

## Description



On C2xxx, C5xxx, or C6xxx processors, this block directs the TI compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

The block does not verify whether the entries for your variable are valid, such as checking the variable name, data type, or section. You must ensure that all variable names are valid, that they use valid data types, and that all section names you specify are valid as well.

The block does not have input or output ports. It only allocates a memory location. You do not connect it to other blocks in your model.

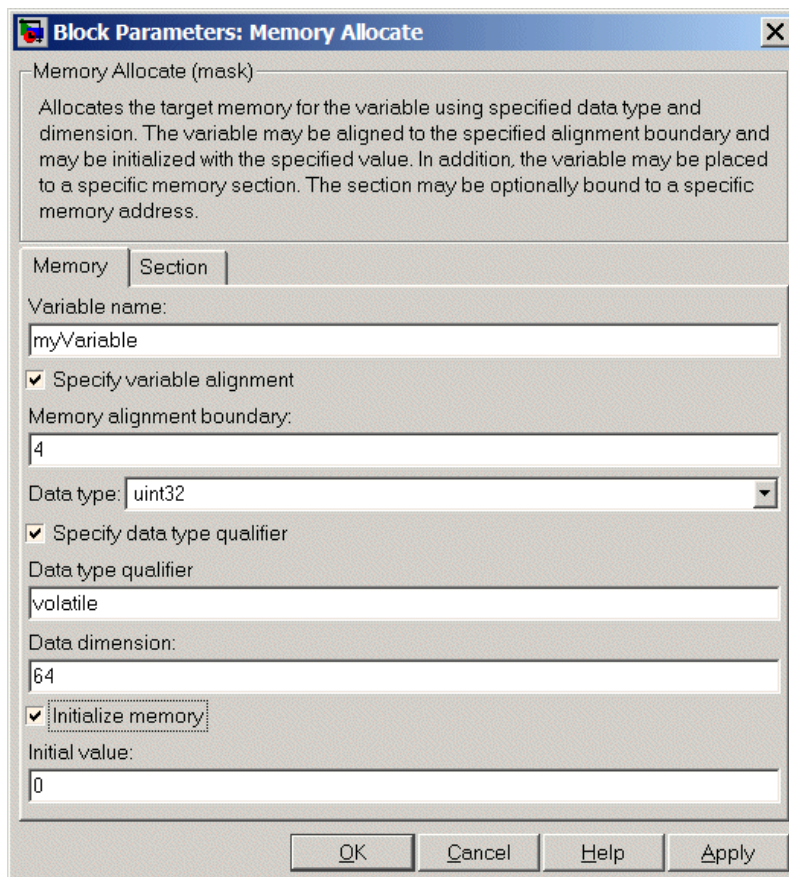
## Dialog Box

The block dialog box comprises multiple tabs:

- **Memory** — Allocate the memory for storing variables. Specify the data type and size.
- **Section** — Specify the memory section in which to allocate the variable.

The dialog box images show all of the available parameters enabled. Some of the parameters shown do not appear until you select one or more other parameters.

# Memory Allocate



The following sections describe the contents of each pane in the dialog box.

## Memory Parameters

**Block Parameters: Memory Allocate**

Memory Allocate (mask)

Allocates the target memory for the variable using specified data type and dimension. The variable may be aligned to the specified alignment boundary and may be initialized with the specified value. In addition, the variable may be placed to a specific memory section. The section may be optionally bound to a specific memory address.

Memory | Section

Variable name:  
myVariable

Specify variable alignment

Memory alignment boundary:  
4

Data type: uint32

Specify data type qualifier

Data type qualifier  
volatile

Data dimension:  
64

Initialize memory

Initial value:  
0

OK Cancel Help Apply

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.

# Memory Allocate

---

## **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

**Block Parameters: Memory Allocate**

Memory Allocate (mask)

Allocates the target memory for the variable using specified data type and dimension. The variable may be aligned to the specified alignment boundary and may be initialized with the specified value. In addition, the variable may be placed to a specific memory section. The section may be optionally bound to a specific memory address.

Memory Section

Specify memory section

Memory section:

mySEC1

Bind memory section

Section start address:

hex2dec('8000')

OK Cancel Help Apply

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

# Memory Allocate

---

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**.

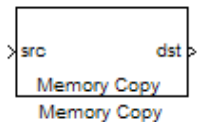
## See Also

Memory Copy

**Purpose** Copy to and from memory section

**Library** Block Library: idelinklib\_common

## Description



In generated code, this block copies variables or data from and to processor memory as configured by the block parameters. Your model can contain as many of these blocks as you require to manipulate memory on your processor.

Each block works with one variable, address, or set of addresses provided to the block. Parameters for the block let you specify both the source and destination for the memory copy, as well as options for initializing the memory locations.

Using parameters provided by the block, you can change options like the memory stride and offset at run time. In addition, by selecting various parameters in the block, you can write to memory at program initialization, at program termination, and at every sample time. The initialization process occurs once, rather than occurring for every read and write operation.

With the custom source code options, the block enables you to add custom ANSI C source code before and after each memory read and write (copy) operation. You can use the custom code capability to lock and unlock registers before and after accessing them. For example, some processors have registers that you may need to unlock and lock with EALLOW and EDIS macros before and after your program accesses them.

If your processor or board supports quick direct memory access (QDMA) the block provides a parameter to check that implements the QDMA copy operation, and enables you to specify a function call that can indicate that the QDMA copy is finished. Only the C621x, C64xx, and C671x processor families support QDMA copy.

## 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

# Memory Copy

---

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.

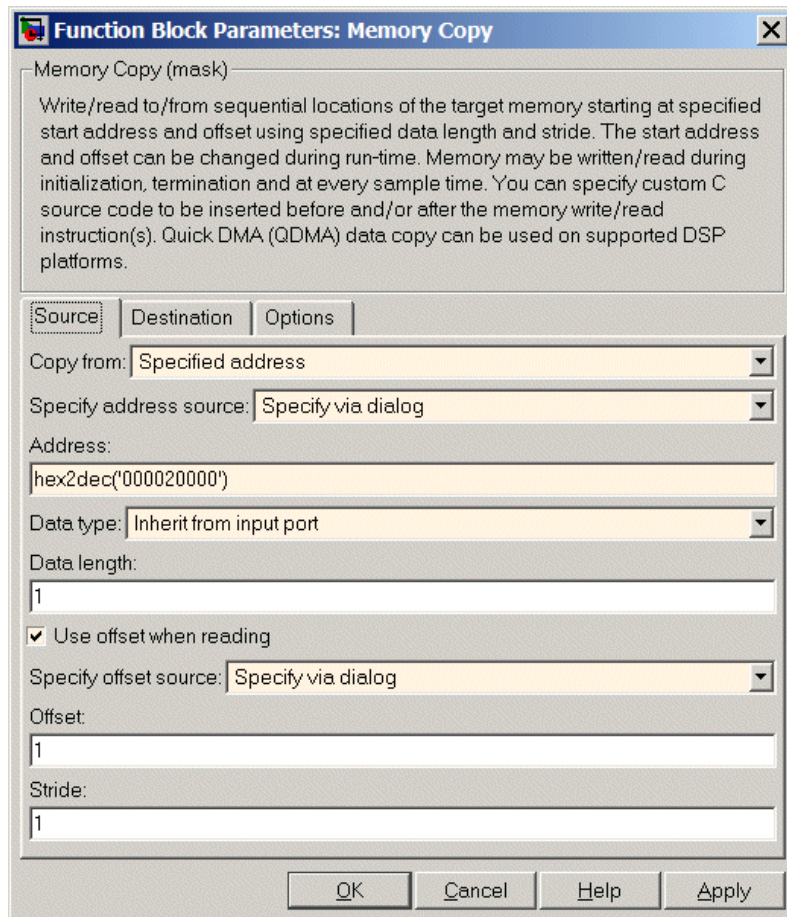
## Dialog Box

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.

# Memory Copy

## Source Parameters

Function Block Parameters: Memory Copy

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source | Destination | Options

Copy from: Specified address

Specify address source: Specify via dialog

Address:  
hex2dec('000020000')

Data type: Inherit from input port

Data length:  
1

Use offset when reading

Specify offset source: Specify via dialog

Offset:  
1

Stride:  
1

OK Cancel Help Apply

### Copy from

Select the source of the data to copy. Choose one of the entries on the list:

- **Input port** — This source reads the data from the block input port.

- Specified address — This source reads the data at the specified location in **Specify address source** and **Address**.
- Specified source code symbol — This source tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

---

**Note** If you do not select **Input port** for **Copy from**, change **Data type** from the default **Inherit from source** to one of the data types on the **Data type** list. If you do not make the change, you receive an error message that the data type cannot be inherited because the input port does not exist.

---

Depending on the choice you make for **Copy from**, you see other parameters that let you configure the source of the data to copy.

### **Specify address source**

This parameter directs the block to get the address for the variable either from an entry in **Address** or from the input port to the block. Select either **Specify via dialog** or **Input port** from the list. Selecting **Specify via dialog** activates the **Address** parameter for you to enter the address for the variable.

When you select **Input port**, the port label on the block changes to **&src**, indicating that the block expects the address to come from the input port. Being able to change the address dynamically lets you use the block to copy different variables by providing the variable address from an upstream block in your model.

### **Source code symbol**

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax. Enter a string to specify the symbol exactly as you use it in your code.

# Memory Copy

---

## 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.

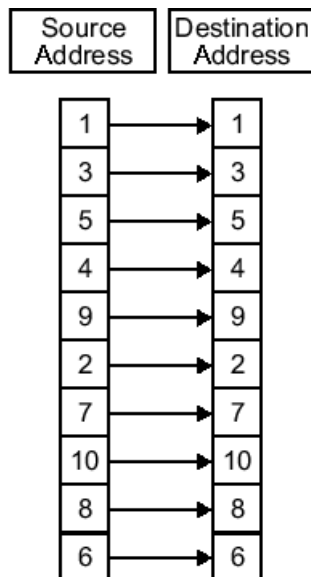
## Stride

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.

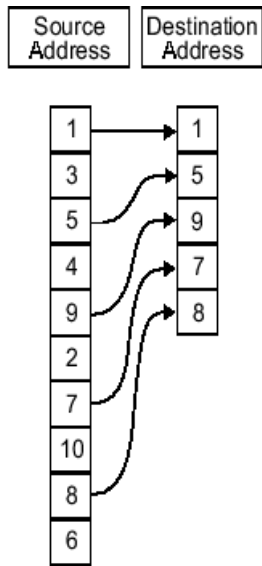
The next two figures help explain the stride concept. In the first figure you see data copied without any stride. Following that figure, the second figure shows a stride value of two applied to reading the input when the block is copying the input to an output location. You can specify a stride value for the output with parameter **Stride** on the **Destination** pane. Compare stride with offset to see the differences.

# Memory Copy

---



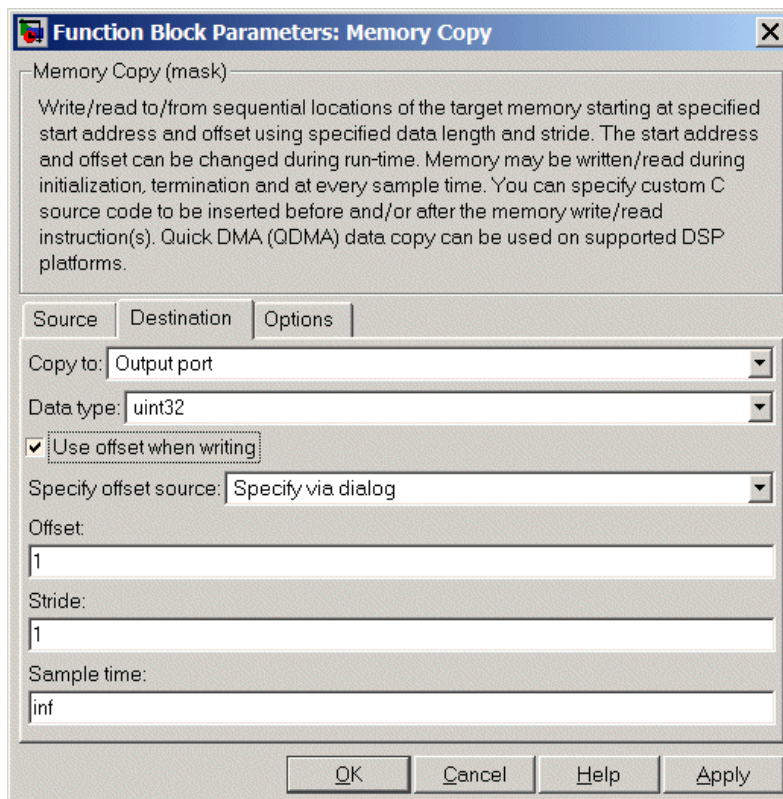
Input Stride = 1  
Output Stride = 1  
Number of Elements Copied = 10



Input Stride = 2  
Output Stride = 1  
Number of Elements Copied = 5

# Memory Copy

## 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');
```

# Memory Copy

---

For this example, you could enter either 8192 or `hex2dec('2000')` as the address.

## Data type

Use this parameter to specify the type of data that your variable uses. The list includes the supported data types, such as `int8`, `uint32`, and `Boolean`, and the option `inherit` from source for inheriting the data type for the variable from the block input port.

## Specify offset source

The block provides two sources for the offset—`Input port` and `Specify via dialog`. Selecting `Input port` configures the block input to read the offset value by adding an input port labeled `src ofs`. This port enables your program to change the offset dynamically during execution by providing the offset value as an input to the block. If you select `Specify via dialog`, you enable the **Offset** parameter in this dialog box so you can enter the offset to use when writing the output data.

## Offset

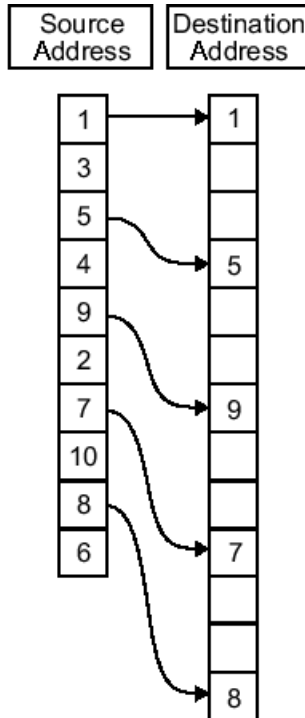
**Offset** tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

## Stride

Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially, but by skipping addresses equal to the stride. **Stride** must be a positive integer.

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in

the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.



Input Stride = 2  
Output Stride = 3  
Number of Elements Copied = 5

## Sample time

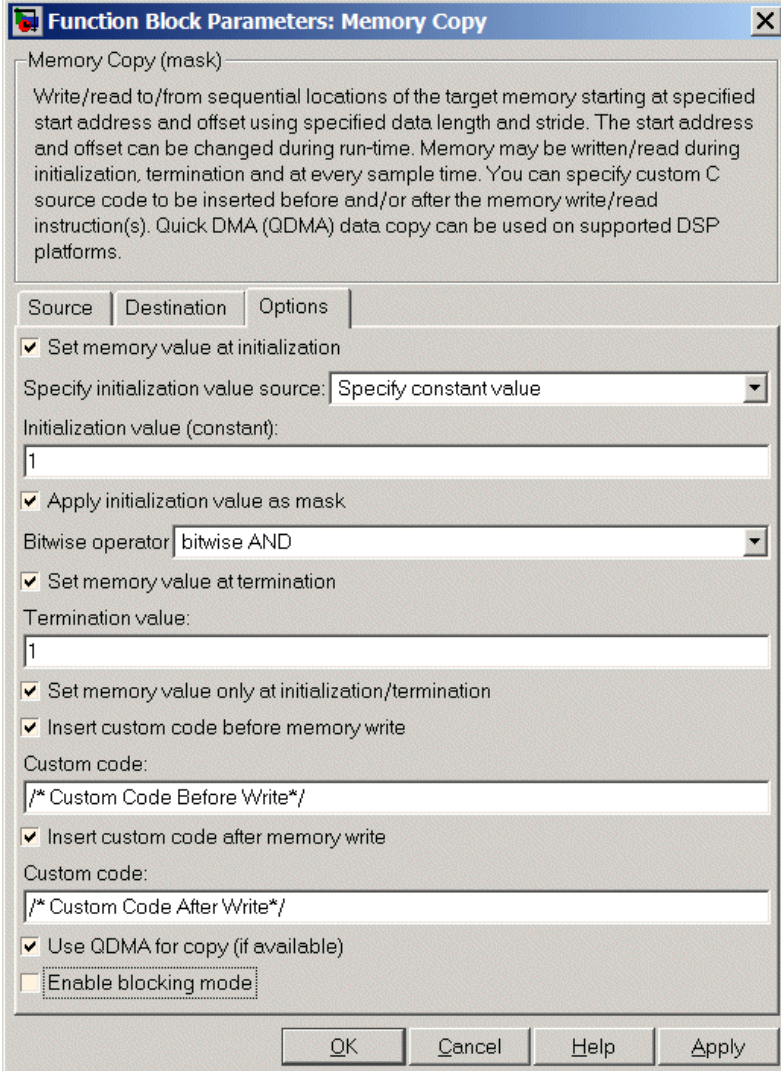
**Sample time** sets the rate at which the memory copy operation occurs, in seconds. The default value `Inf` tells the block to use a constant sample time. You can set **Sample time** to `-1` to direct the block to inherit the sample time from the input, if there is one,

# Memory Copy

---

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



**Function Block Parameters: Memory Copy**

Memory Copy (mask)

Write/read to/from sequential locations of the target memory starting at specified start address and offset using specified data length and stride. The start address and offset can be changed during run-time. Memory may be written/read during initialization, termination and at every sample time. You can specify custom C source code to be inserted before and/or after the memory write/read instruction(s). Quick DMA (QDMA) data copy can be used on supported DSP platforms.

Source | Destination | Options

Set memory value at initialization

Specify initialization value source: Specify constant value

Initialization value (constant):

1

Apply initialization value as mask

Bitwise operator: bitwise AND

Set memory value at termination

Termination value:

1

Set memory value only at initialization/termination

Insert custom code before memory write

Custom code:

/\* Custom Code Before Write\*/

Insert custom code after memory write

Custom code:

/\* Custom Code After Write\*/

Use QDMA for copy (if available)

Enable blocking mode

OK Cancel Help Apply

## **Set memory value at initialization**

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you select this option, use the **Set memory value at termination** and **Specify initialization value source** parameters to set your desired value. Alternately, you can tell the block to get the initial value from the block input.

## **Specify initialization value source**

After you check **Set memory value at initialization**, use this parameter to select the source of the initial value. Choose either

- **Specify constant value** — Sets a single value to use when your program initializes memory. 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.

# Memory Copy

---

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

# Target Preferences/Custom Board

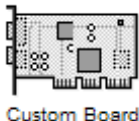
---

**Purpose** Configure model for a supported processor

**Library**

Block library: idelinklib\_adivdsp  
Block library: idelinklib\_ghsmulti  
Block library: idelinklib\_ticcs  
Block library: idelinklib\_eclipseide

## Description



Use this block to configure hardware settings and code generation features for your custom board. Include this block in models you use to generate Real-Time Workshop code to run on processors and boards. It does not connect to any other blocks, but stands alone to set the processor preferences for the model.

---

**Tip** Place only one Target Preferences block in your model.

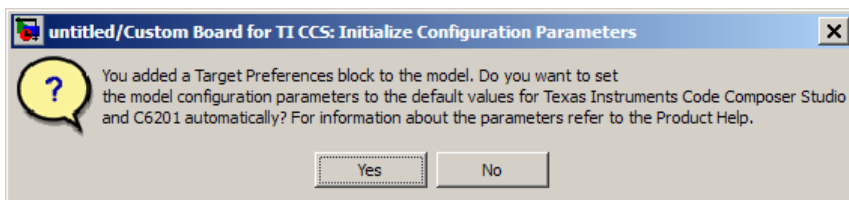
When you are generating code for a model, place the Target Preferences block at the top level of your model.

When you are generating code for a subsystem, place the Target Preferences block at the subsystem level of your model.

---

Setting the Target Preference block options identifies your processor and board to Real-Time Workshop software, Embedded IDE Link, and Simulink software. Setting the options also configures the memory map for your processor.

Click **Yes** when you drag and drop the Target Preferences/Custom Board block to your model and you get a prompt to “set the model configuration parameter to the default values”? For example:



If you select **No**, the settings may be incorrect. If you build the model with the incorrect settings, the software generates error messages.

## Generating Code from Model Subsystems

Real-Time Workshop software provides the ability to generate code from a selected subsystem in a model. To generate code for custom hardware from a subsystem, the subsystem model must include a Target Preferences block.

## Dialog Boxes

This reference page section contains the following subsections:

- “Board Pane” on page 7-32
- “Memory Pane” on page 7-37
- “Sections Pane” on page 7-40
- “DSP/BIOS Pane” on page 7-43
- “Peripherals Pane” on page 7-47
- “Add Processor Dialog Box” on page 7-96

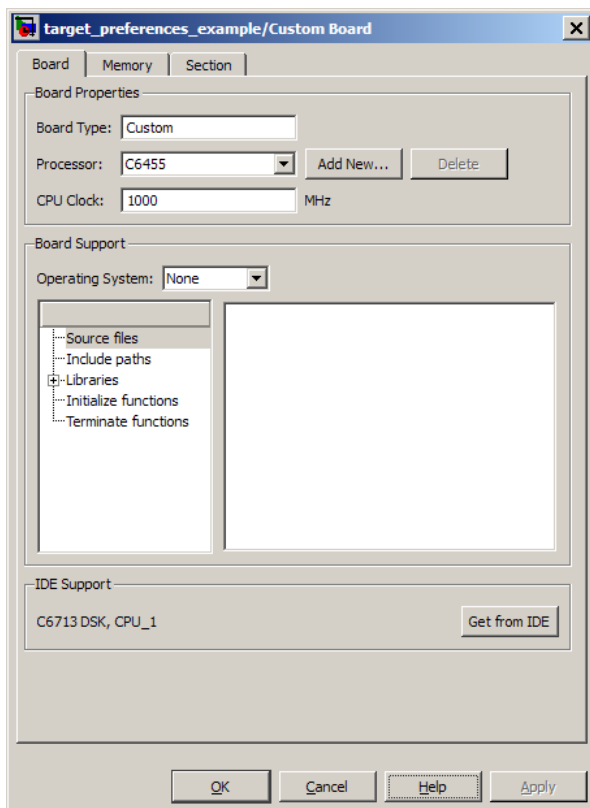
Target Preferences block dialog boxes provide tabbed access to the following panes with options you set for the processor and board:

- Board Pane — Select the processor, set the clock speed, and identify the processor. In addition, **Add new** on this pane opens the New Processor dialog box.
- Memory Pane — Set the memory allocation and layout on the processor (memory mapping).

# Target Preferences/Custom Board

- Sections Pane — Determine the arrangement and location of the sections on the processor and compiler information.
- DSP/BIOS Pane — With Texas Instruments CCS IDE and C6000 processors: Specify how to configure tasking features of DSP/BIOS.
- Peripherals Pane — With 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.

## Board Pane



The following options appear on the **Board** pane, under the **Board Properties**, **Board Support**, and **IDE Support** labels.

## **Board type**

Enter the type of your target board. Enter **Custom** to support any board that uses a processor on the **Processor** list, or enter the name of a supported board. If you are using one of the explicitly supported boards, choose the Target Preferences/Custom Board block for that board from the Simulink .

## **Processor**

Select the target processor type. This action applies default values to many of the remaining settings, such as those under the **Memory** and **Section** tabs.

If the Embedded IDE Link product supports an operating system for the processor, it enables the **Operating system** option.

## **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 7-96.

## **Delete**

Delete a processor that you added to the **Processor** list. You cannot delete processors that you did not add.

## **CPU Clock (MHz)**

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting the actual clock rate produces 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

# Target Preferences/Custom Board

---

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:

$$100,000,000/1000 = 1 \text{ Sine block interrupt per } 100,000 \text{ clock ticks}$$

Thus, report the correct clock rate, or the interrupts come at the wrong times and the results are incorrect.

## 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 the IDE installation directory.

```
$(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.

**Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

## Operating System

If you select a processor for which the Embedded IDE Link product provides support, Embedded IDE Link enables the **Operating system** option. You can select an operating system for your model.

For more information about using Eclipse IDE to run target applications on Windows or Linux, see “Supported Operating Systems”.

For more information about using CCS IDE to run target applications on DSP/BIOS, see “Targeting with DSP/BIOS Options”.

## Get from IDE

Import the boards and processors defined in the Code Composer Studio and VisualDSP++ IDEs. This information populates the **Board name** and **Processor name** options. Click the **Apply** button to make this information available in the other options.

## Target Preferences/Custom Board

---

This feature does not work if you do not have an IDE or your IDE does not support this feature.

### **Board name**

**Board name** appears after you click **Get from IDE**. Select the board you are using. Match **Board name** with the **Board Type** 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.

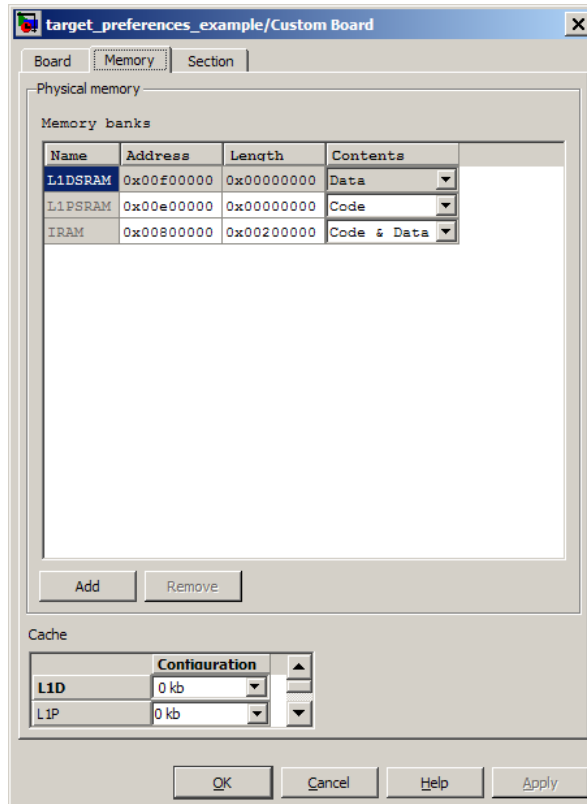
---

**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

# Target Preferences/Custom Board

---

- **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.

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 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 are in hexadecimal format and 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).

## 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 an L2 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.

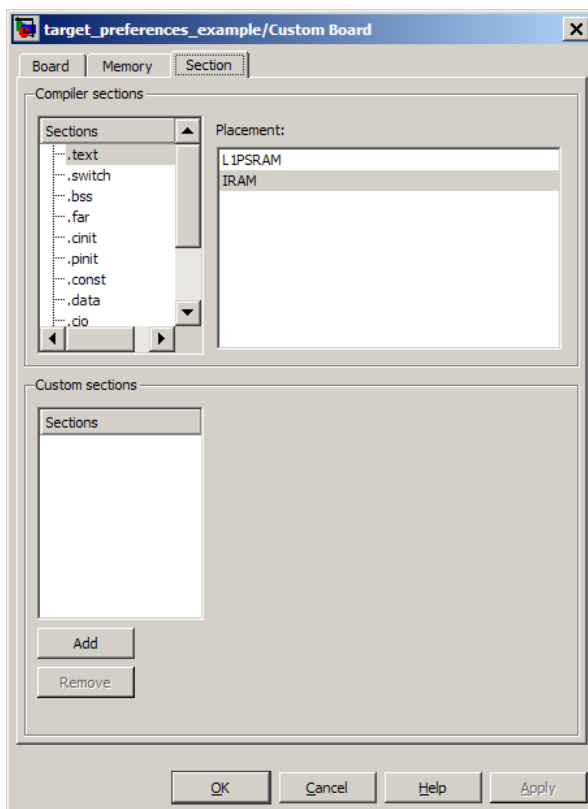
If your processor supports the two-level memory scheme, this option enables the L2 cache on the processor.

# Target Preferences/Custom Board

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.

## Sections Pane



Options on this pane specify where program sections go in memory. Program sections are distinct from memory segments—sections are

# Target Preferences/Custom Board

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 Sections 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.

<b>String</b>	<b>Section List</b>	<b>Description of the Section Contents</b>
<code>.bss</code>	Compiler	Static and global C variables in the code
<code>.cinit</code>	Compiler	Tables for initializing global and static variables and constants
<code>.cio</code>	Compiler	Standard I/O buffer for C programs
<code>.const</code>	Compiler	Data defined with the C qualifier and string constants
<code>.data</code>	Compiler	Program data for execution
<code>.far</code>	Compiler	Variables, both static and global, defined as far variables
<code>.pinit</code>	Compiler	Load allocation of the table of global object constructors section
<code>.stack</code>	Compiler	The global stack
<code>.switch</code>	Compiler	Jump tables for switch statements in the executable code

# Target Preferences/Custom Board

---

String	Section List	Description of the Section Contents
.system	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants

You can learn more about memory sections and objects in the online help for your IDE.

## Default Sections

When you highlight a section on the list, **Description** show a brief description of the section. Also, **Placement** shows you the memory allocation of the section.

## Description

Provides a brief explanation of the contents of the selected entry on the **Compiler sections** list.

## Placement

Shows the allocation of the selected **Compiler sections** entry in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments as defined in the physical memory map on the **Memory** pane. Select one of the listed memory segments to allocate the highlighted compiler section to the segment.

To see a description of the placement item, hover your mouse pointer over the item for a few moments.

## Custom Sections

If your program uses code or data sections that are not in the **Compiler sections**, add the new sections to **Custom sections**.

## Sections

This window lists data sections that are not in the **Compiler sections**.

## Placement

With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select 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**.

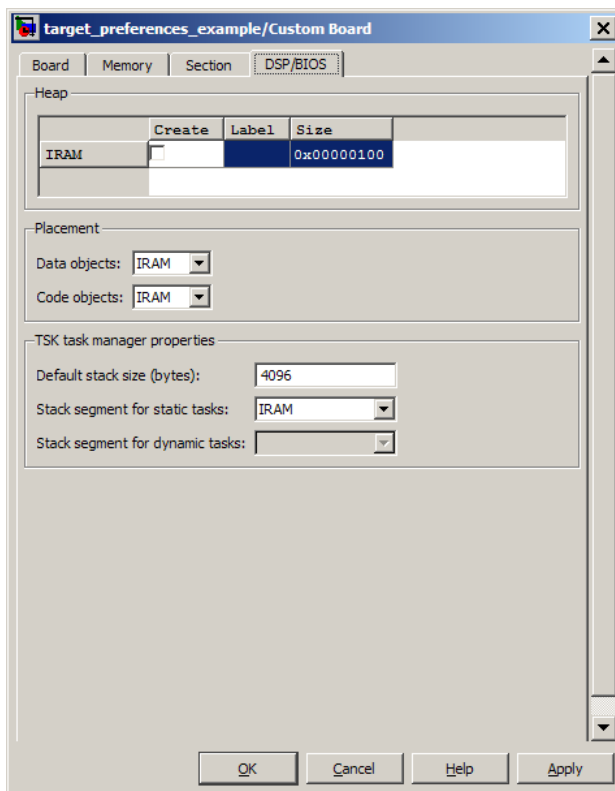
## DSP/BIOS Pane

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.

# Target Preferences/Custom Board

---



Selecting DSP/BIOS for **Operating system** on the Board Info 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 Info** 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** is not available 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.

# Target Preferences/Custom Board

---

## **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.

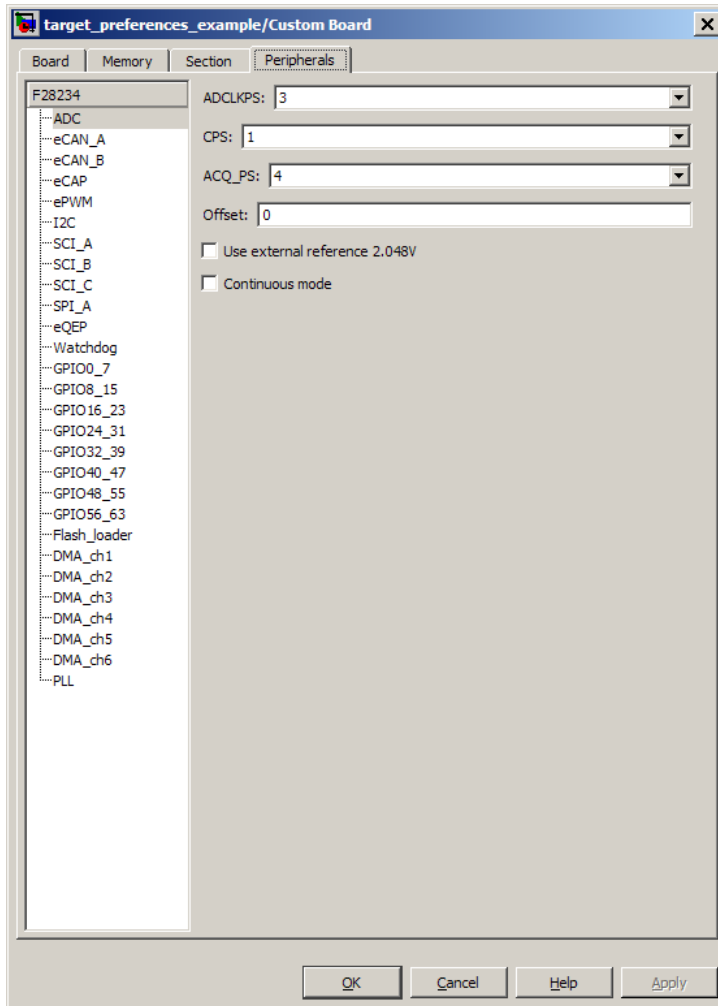
The list offers IDRAM for locating the stack in memory. The Memory pane provides more options for the physical memory on the processor.

## **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 7-37.

# Target Preferences/Custom Board

## 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.

## Target Preferences/Custom Board

---

You must have Target Support Package installed to enable this pane when you select a C2000 processor.

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.

<b>Peripheral Name</b>	<b>Description</b>
ADC	Report the settings for the Analog-to-Digital Converter
eCAN_A	Report or set the enhanced Controller Area Network parameters for module A
eCAN_B	Report or set the enhanced Controller Area Network parameters for module B
eCAP	Report or assign enhanced CAPture module pins to general purpose IO pins
ePWM	Report or assign enhanced Pulse Width Modulation pins to general purpose IO pins
I2C	Report or set Inter-Integrated Circuit parameters
SCI_A	Report or set the Serial Communications Interface parameters for module A
SCI_B	Report or set the Serial Communications Interface parameters for module B
SCI_C	Report or set the Serial Communications Interface parameters for module C
SPI_A	Report or set the Serial Peripheral Interface parameters for module A
SPI_B	Report or set the Serial Peripheral Interface parameters for module B

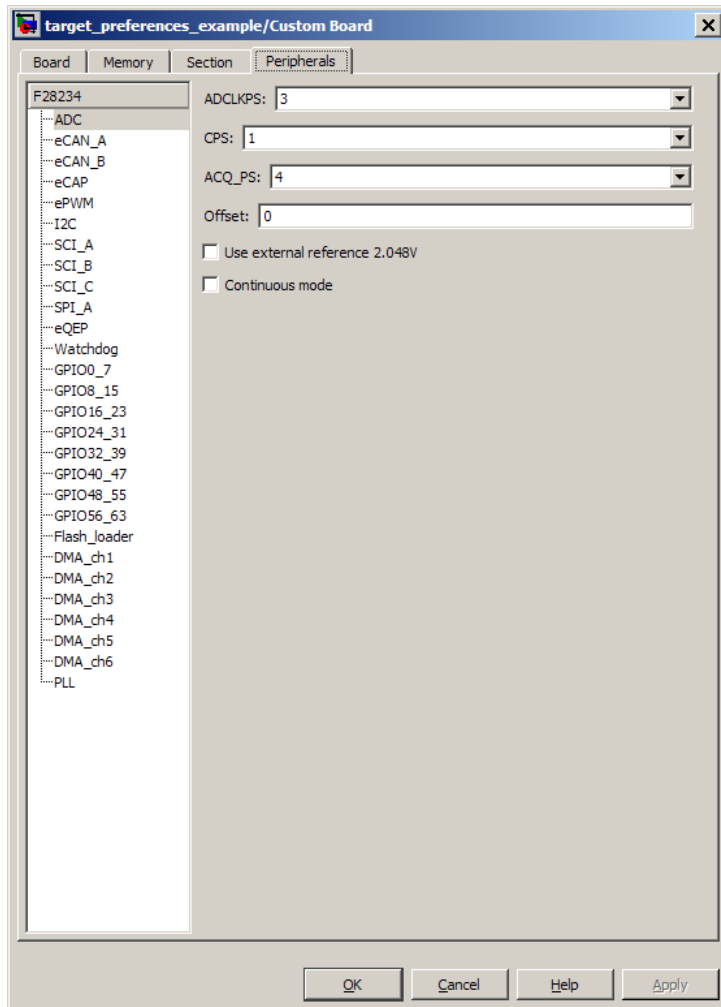
## Target Preferences/Custom Board

---

Peripheral Name	Description
SPI_C	Report or set the Serial Peripheral Interface parameters for module C
SPI_D	Report or set the Serial Peripheral Interface parameters for module D
eQEP	Report or assign enhanced Quadrature Encoder Pulse module pins to general purpose IO pins
Watchdog	Enable and configure timing for watchdog module
GPIO[pin#]	Configure input qualification types for General Purpose Input Output pins
Flash_loader	Enable and configure flash memory programmer. Manually program flash
DMA_ch[#]	Enable and configure Direct Memory Access channels
PLL	Adjust clock settings to match custom oscillator frequencies

# Target Preferences/Custom Board

## ADC



The high-speed peripheral clock (HSPCLK) controls the internal timing of the ADC module. The ADC derives the operating clock speed from the HSPCLK speed in several prescaler stages. For more information about configuring these scalars, refer to “Configuring ADC

Parameters for Acquisition Window Width” in the Target Support Package documentation.

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

# Target Preferences/Custom Board

---

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.

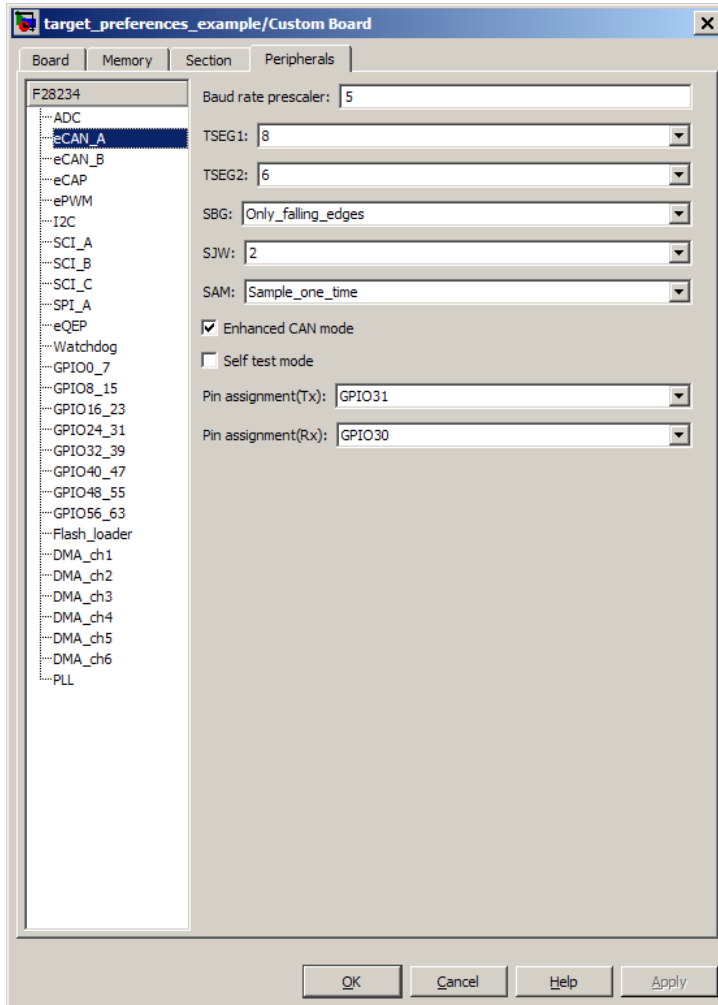
## **XINT2SOC**

(For Piccolo processors) Select the pin to which the ADC sends the XINT2SOC pulse.



# Target Preferences/Custom Board

## eCAN\_A, eCAN\_B



For more help on setting the timing parameters for the eCAN modules, refer to [Configuring Timing Parameters for CAN Blocks](#). You can set the following parameters for the eCAN module:

# Target Preferences/Custom Board

---

## **Baud rate prescaler**

Value by which to scale the bit rate. Valid values are from 1 to 256.

## **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)**

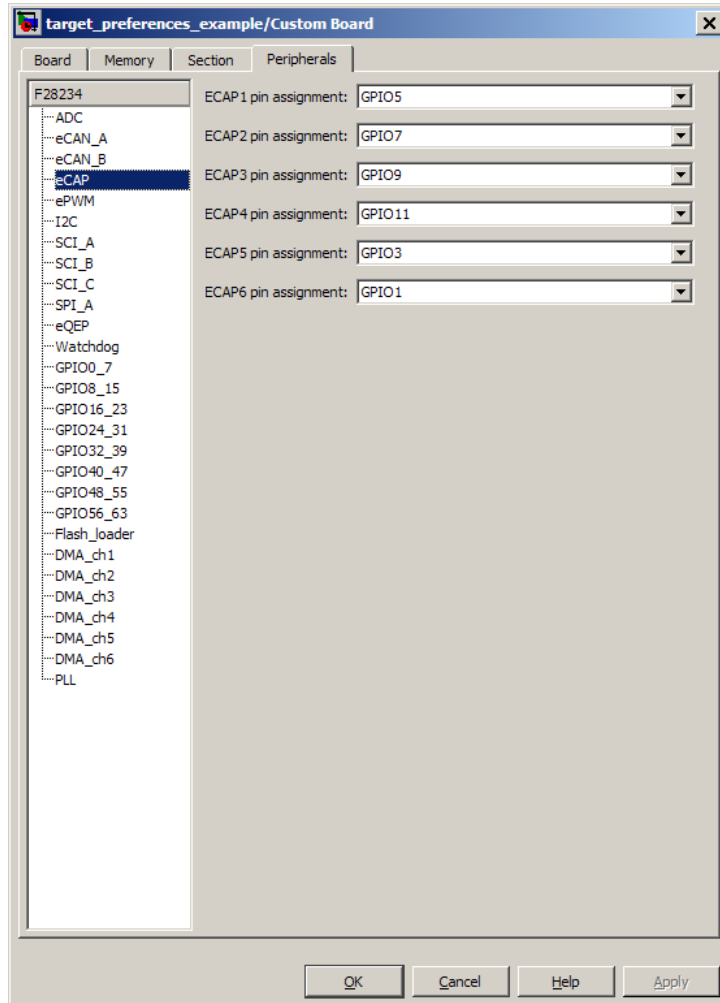
Assigns the CAN receive pin to use with the `eCAN_B` module. Possible values are `GPI010`, `GPI013`, `GPI017`, and `GPI021`.

# Target Preferences/Custom Board

## Pin assignment (Tx)

Assigns the CAN transmit pin to use with the eCAN\_B module.  
Possible values are GPIO8, GPIO12, GPIO16, and GPIO20.

## eCAP



# Target Preferences/Custom Board

---

Assigns eCAP pins to GPIO pins if required.

**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**

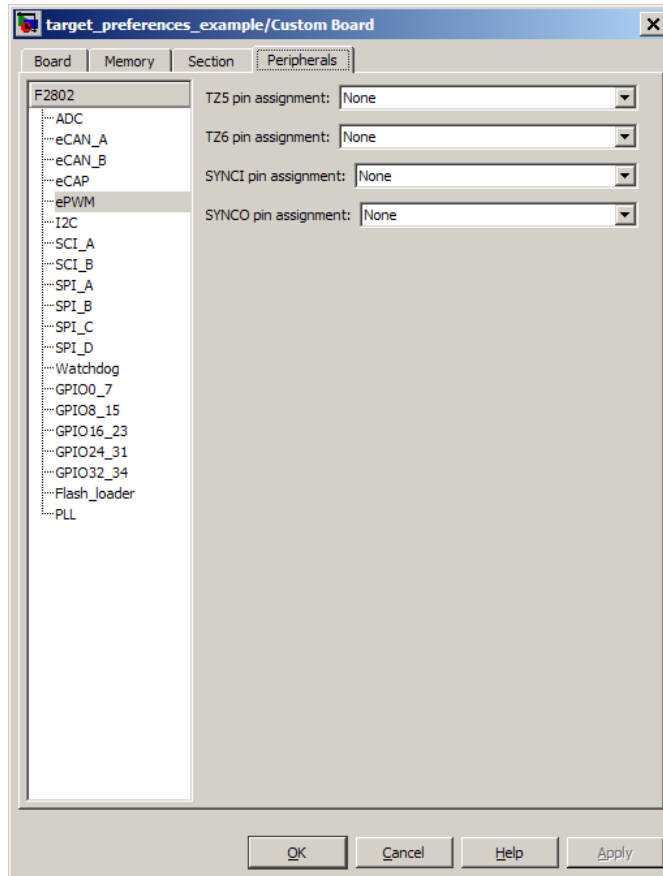
Select an option from the list—None, GPIO9, or GPIO26.

**ECAP4 pin assignment**

Select an option from the list—None, GPIO11, or GPIO27.

# Target Preferences/Custom Board

## ePWM



Assigns ePWM signals to GPIO pins, if required.

### **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.

# Target Preferences/Custom Board

---

## **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**

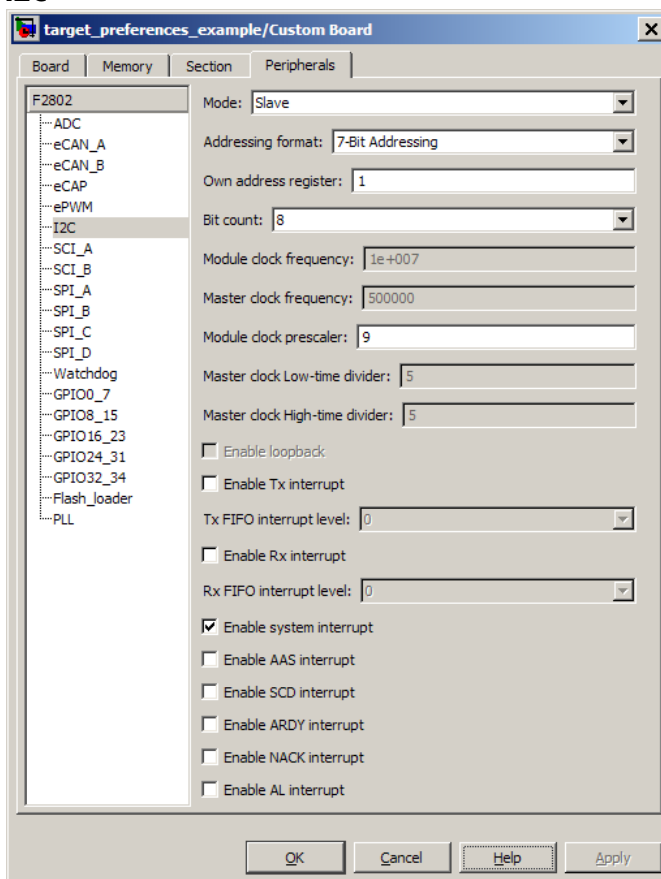
Assigns the trip-zone input 5 (TZ5) to a GPIO pin. Choices are None (the default), GPIO16, and GPIO28.

## **TZ6 pin assignment**

Assigns the trip-zone input 6 (TZ6) to a GPIO pin. Choices are None (the default), GPIO17, and GPIO29.

# Target Preferences/Custom Board

## 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.

# Target Preferences/Custom Board

---

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 from 0–255.

$$\text{Module clock frequency} = \text{I2C input clock frequency} / (\text{Module clock prescaler} + 1)$$

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.

# Target Preferences/Custom Board

---

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 =  $T_{\text{mod}} \times (\text{ICCL} + d)$ .

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x 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 =  $T_{\text{mod}} \times (\text{ICCL} + d)$ .

For more information about this value, consult the “Formula for the Master Clock Period” section in the *TMS320x280x 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.

## Target Preferences/Custom Board

---

When enabled, the I2C module generates an interrupt (AAS bit = 1) upon receiving one of the following:

- Its **Own address register**
- A general call (all zeros)
- A data byte is in free data format

When enabled, the I2C module clears the interrupt (AAS = 0) upon receiving one of the following:

- Multiple START conditions (7-bit addressing mode only)
- A slave address that is different from **Own address register** (10-bit addressing mode only)
- A NACK or a STOP condition

This parameter corresponds to bit 6 (AAS) of the Interrupt Enable Register (I2CIER).

### **Enable SCD interrupt**

Enable stop condition detected interrupt.

When enabled, the I2C module generates an interrupt (SCD bit = 1) when the CPU detects a stop condition on the I2C bus.

When enabled, the I2C module clears the interrupt (SCD = 0) upon one of the following events:

- The CPU reads the I2CISRC while it indicates a stop condition
- A reset of the I2C module
- Someone manually clears the interrupt

This parameter corresponds to bit 5 (SCD) of the Interrupt Enable Register (I2CIER).

### **Enable ARDY interrupt**

Enable register-access-ready interrupt enable bit.

When enabled, the I2C module generates an interrupt (ARDY bit = 1) when the previous address, data, and command values in the I2C module registers have been used and new values can be written to the I2C module registers.

This parameter corresponds to bit 2 (ARDY) of the Interrupt Enable Register (I2CIER).

### **Enable NACK interrupt**

Enable no-acknowledgment interrupt enable bit.

When enabled, the I2C module generates an interrupt (NACK bit = 1) when the module is a transmitter in master or slave mode and it receives a NACK condition.

This parameter corresponds to bit 1 (NACK) of the Interrupt Enable Register (I2CIER).

### **Enable AL interrupt**

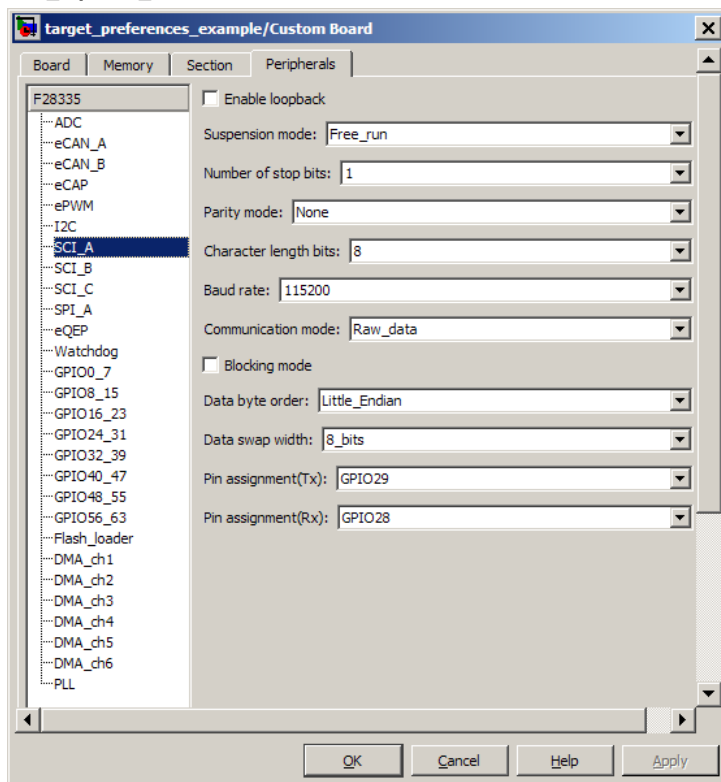
Enable arbitration-lost interrupt.

When enabled, the I2C module generates an interrupt (AL bit = 1) when the I2C module is operating as a master transmitter and loses an arbitration contest with another master transmitter.

This parameter corresponds to bit 0 (AL) of the Interrupt Enable Register (I2CIER).

# Target Preferences/Custom Board

## SCI\_A, SCI\_B



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

# Target Preferences/Custom Board

---

---

**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.

## **Data swap width**

Select 8\_bits or 16\_bits. 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



# Target Preferences/Custom Board

sequence is complete. Free\_run continues running regardless of the breakpoint.

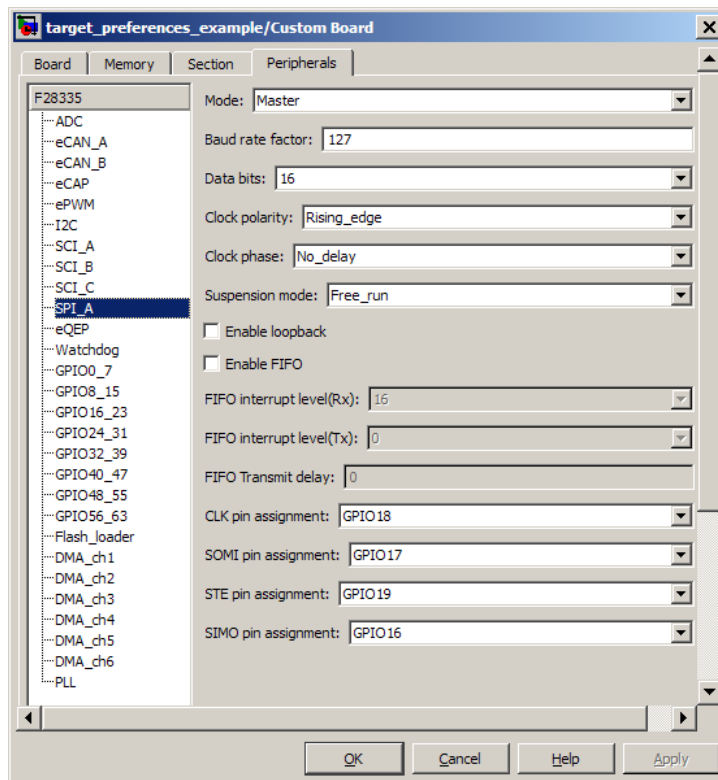
## Pin assignment (Rx)

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:

# Target Preferences/Custom Board

---

## **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 3 pins instead of the normal 4 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**

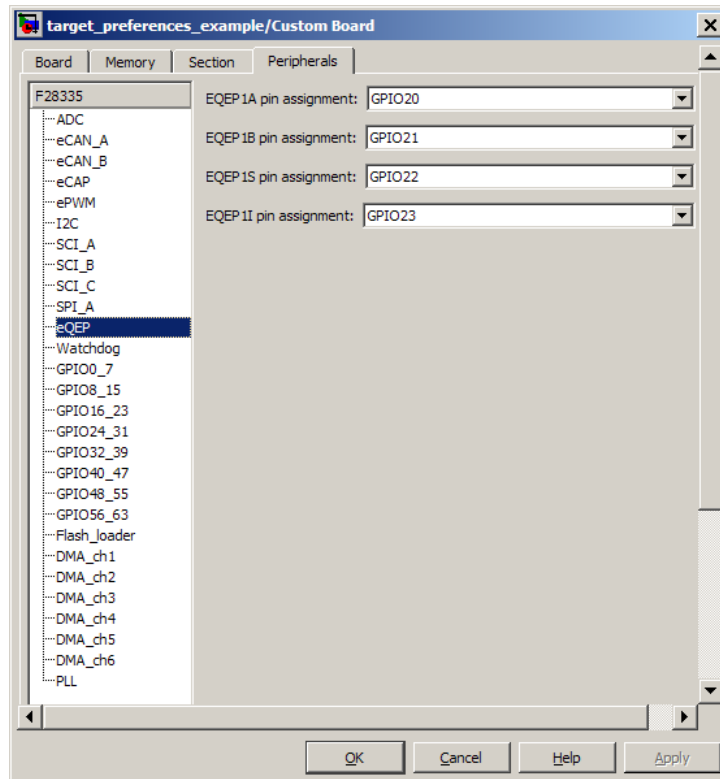
Assigns the SPI something (STE) to a GPIO pin. Choices are None (default), GPIO15, or GPIO27.

**SIMO pin assignment**

Assigns the SPI something (SIMO) to a GPIO pin. Choices are None (default), GPIO12, or GPIO24.

# Target Preferences/Custom Board

## 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

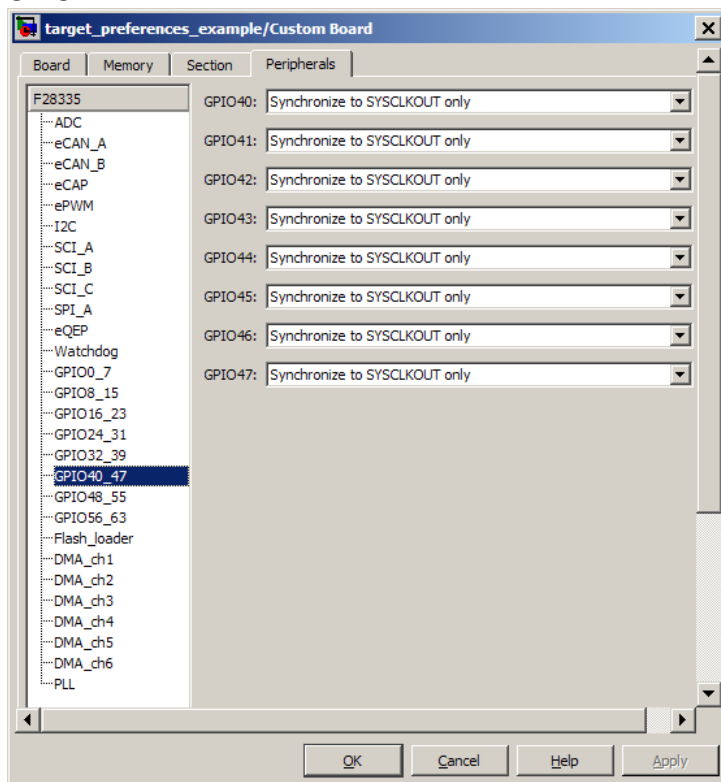
Configure the watchdog to reset the processor or generate an interrupt when the software fails to reset the watchdog counter:

- 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.

# Target Preferences/Custom Board

This parameter corresponds to bit 1 (WDENINT) of the System Control and Status Register (SCSR).

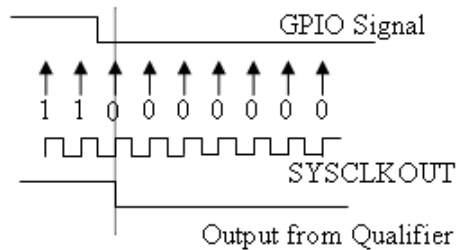
## GPIO



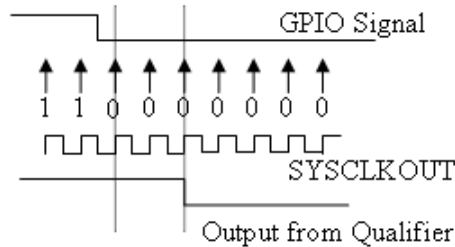
Each pin selected for input offers three signal qualification types:

- 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.

# Target Preferences/Custom Board

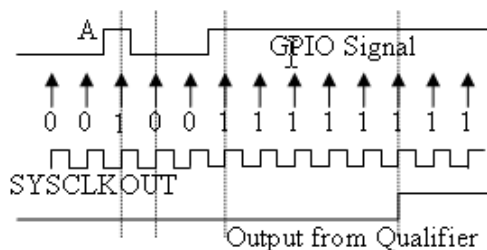


- **Qualification using 3 samples** — This setting requires three consecutive cycles of the same value for the output value to change. The following figure shows that, in the third cycle, the GPIO value changes to 0, but the qualifier output is still 1 because it waits for three consecutive cycles of 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.

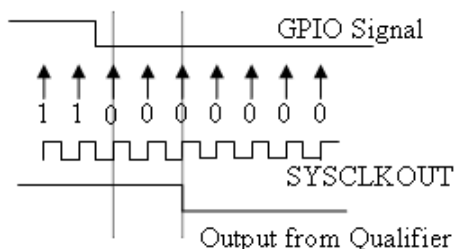
# Target Preferences/Custom Board



## 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  $\text{SYSCLKOUT}/(2 * \text{Prescaler})$ , except for zero. When **Qualification sampling period prescaler=0**, a sample is taken every SYSCLKOUT clock tick. For example, a prescale setting of 0 means that a sample is taken on each SYSCLKOUT tick.

The following figure shows the SYSCLKOUT ticks, a sample taken every clock tick, and the **Qualification type** set to Qualification using 3 samples. In this case, the **Qualification sampling period prescaler=0**:

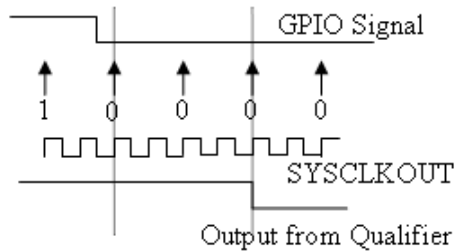


In the next figure **Qualification sampling period prescaler=1**. A sample is taken every two clock ticks, and the **Qualification type** is

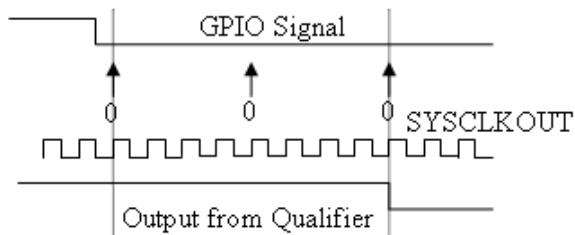


# Target Preferences/Custom Board

set to **Qualification** using 3 samples. The output signal changes much later than if **Qualification sampling period prescaler=0**.

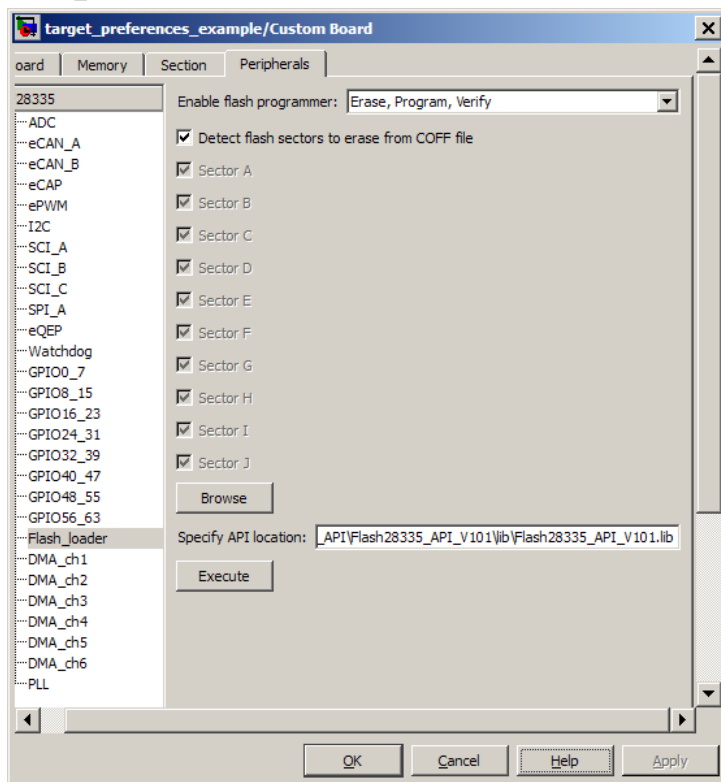


In the following figure, **Qualification sampling period prescaler=2**. Thus, a sample is taken every four clock ticks, and the **Qualification type** is set to **Qualification** using 3 samples.



# Target Preferences/Custom Board

## 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 in the *Target Support Package User’s Guide* 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.

## **Erase Sector Selection**

When **Detect Flash sectors to erase from COFF file** is disabled, the selected flash sectors are erased.

## **Specify API location**

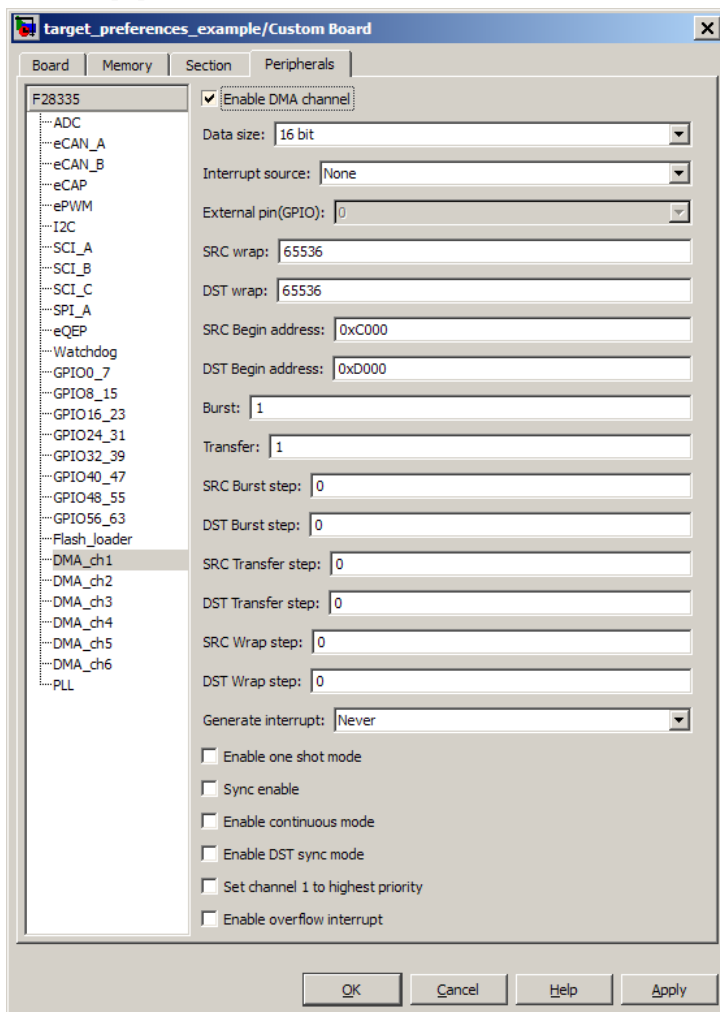
Specify the directory path of the TI flash API executable you downloaded and installed on your computer. Use **Browse** to locate the file or enter the path in the text box.

## **Execute**

Click this button to initiate the task selected in **Enable Flash Programmer**.

# Target Preferences/Custom Board

## DMA\_ch[#]



The Direct Memory Access module transfers data directly between peripherals and memory using a dedicated bus, increasing overall system performance.

You can individually enable and configure each DMA channel.

The DMA module services are event driven. Using the **Interrupt source** and **External pin (GPIO)** parameters, you can configure a wide range of peripheral interrupt event triggers.

To 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 C280x/C28x3x 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:

# Target Preferences/Custom Board

---

- 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 C280x/C28x3x 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 C280x/C28x3x 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:

# Target Preferences/Custom Board

---

- *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 GPIO Digital Input and C280x/C2802x/C2803x/C28x3x GPIO Digital Output block reference sections in the *Target Support Package User's Guide*.

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)** 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.

# Target Preferences/Custom Board

---

## **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)** 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)** 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)** 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)** 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,  $\text{dstAdd} = \text{ADCbufadr} - \text{NoC}$ .

- If the target is F28232 or F28332,  $\text{ADCbufadr} = 57340$  (0xDFFC)
- Otherwise,  $\text{ADCbufadr} = 65532$  (0xFFFFC)

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).

---

# Target Preferences/Custom Board

---

## 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)** 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).  $\text{Burst} = \text{NOC} * \text{CVM}$

If **Conversion mode** is Sequential, CVM = 1. If **Conversion mode** is Simultaneous, CVM = 2.

For example, Burst = 6 if NOC = 3 and CVM = 2 ( $6 = 3 * 2$ ).

For more information, see C280x/C28x3x ADC.

---

## 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)** 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)** 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).

## Target Preferences/Custom Board

---

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)** 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)** 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)** the C280x/C28x3x ADC block, the value of this destination parameter is 1.

---

# Target Preferences/Custom Board

---

## 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)** 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)** 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)** 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)** 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.

# Target Preferences/Custom Board

---

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)** 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)** 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 (SYNCSSEL) in the Mode Register (MODE).



---

**Note** When you select **Use DMA (with C28x3x)** 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)** 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

## Target Preferences/Custom Board

---

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).

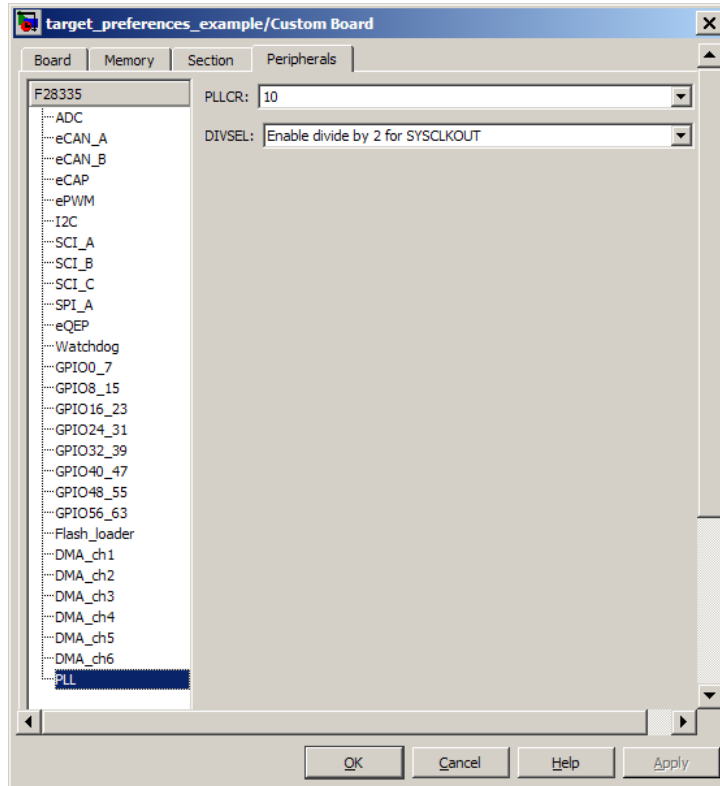
---

**Note** When you select **Use DMA (with C28x3x)** the C280x/C28x3x ADC block, this parameter is disabled.

---

# Target Preferences/Custom Board

## 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.

# Target Preferences/Custom Board

---

Use the following equation to determine the CPU frequency (CLKIN):

$$\text{CLKIN} = (\text{OSCCLK} * \text{PLLCR}) / (\text{DIVSEL or CLKINDIV})$$

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”.

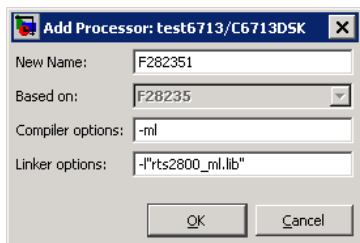
The availability of the DIVSEL or CLKINDIV parameters change depending on the selected processor. If neither parameter is available, use the following equation instead:

$$\text{CLKIN} = (\text{OSCCLK} * \text{PLLCR}) / 1$$

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.

## 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. You can use any valid C string value in this field. 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, Embedded IDE Link 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 `-m1`. 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
C55xx	-m1
C28xx, F28xx, R28xx, F28xxx	-m1

# Target Preferences/Custom Board

---

## **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.

**Purpose**

Custom or demo block

**Description**

This help topic serves as a landing page if you click the help button for a custom or demo block. These blocks are typically undocumented because they are not part of the standard block libraries.

To provide online help for custom blocks you create, see “Providing Your Own Help and Demos”.

# Custom or Demo Block

---



# Function Reference

---

Setup (p. 8-2)	Set up Embedded IDE Link to work with your IDE or toolchain
Constructor (p. 8-3)	Create an object for handling your IDE
File and Project Operations (p. 8-4)	Perform operations on files and projects in your IDE
Processor Operations (p. 8-5)	Perform operations on the processors associated with your IDE
Debug Operations (p. 8-6)	Perform debug operations
Data Manipulation (p. 8-7)	Perform operations on processor memory and registers
Status Operations (p. 8-8)	Get processor status
Grouped by IDE (p. 8-9)	Lists of supported methods for each IDE

## Setup

<code>adivdspsetup</code>	Configure Embedded IDE Link to work with VisualDSP++ IDE
<code>checkEnvSetup</code>	Check system requirements and configure Embedded IDE Link to work with CCS IDE
<code>eclipseidesetup</code>	Configure Embedded IDE Link to work with Eclipse IDE
<code>ghsmulticonfig</code>	Configure Embedded IDE Link to work with MULTI IDE
<code>xmakefilesetup</code>	Configure Embedded IDE Link to generate makefiles

## Constructor

<code>adivdsp</code>	Create handle object to interact with VisualDSP++ IDE
<code>eclipseide</code>	Create handle object to interact with Eclipse IDE
<code>ghsmulti</code>	Create handle object to interact with MULTI IDE
<code>ticcs</code>	Create handle object to interact with CCS IDE

## File and Project Operations

<code>activate</code>	Mark file, project, or build configuration as active
<code>add</code>	Add files to active project in IDE
<code>build</code>	Build or rebuild current project
<code>cd</code>	Set working directory in IDE
<code>close</code>	Close project in IDE window
<code>connect</code>	Connect IDE to processor
<code>dir</code>	Files and directories in current IDE window
<code>getbuildopt</code>	Generate structure of build tools and options
<code>info</code>	Information about processor
<code>list</code>	Information listings from IDE
<code>new</code>	Create project, library, or build configuration in IDE
<code>open</code>	Open project in IDE
<code>remove</code>	Remove file, project, or breakpoint
<code>setbuildopt</code>	Set active configuration build options
<code>symbol</code>	Program symbol table from IDE

## Processor Operations

halt	Halt program execution by processor
load	Load program file onto processor
profile	Generate real-time execution or stack profiling report
reset	Stop program execution and reset processor
restart	Reload most recent program file to processor signal processor
run	Execute program loaded on processor

## **Debug Operations**

animate	Run application on processor to breakpoint
info	Information about processor
insert	Insert debug point in file
remove	Remove file, project, or breakpoint
run	Execute program loaded on processor

## Data Manipulation

address	Memory address and page value of symbol in IDE
pwd	Working directory used by Eclipse
read	Read data from processor memory
regread	Values from processor registers
regwrite	Write data values to registers on processor
write	Write data to processor memory block

## **Status Operations**

`isrunning`

Determine whether processor is  
executing process



## Grouped by IDE

In this section...
“Altium TASKING” on page 8-9
“Analog Devices™ VisualDSP++” on page 8-9
“Eclipse IDE” on page 8-11
“Green Hills® MULTI” on page 8-12
“Texas Instruments Code Composer Studio” on page 8-14

### Altium TASKING

The “Automation Interface” topic in the *Embedded IDE Link User’s Guide* describes the functions and methods for working with Altium TASKING.

### 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 Embedded IDE Link to work with VisualDSP++ IDE
build	Build or rebuild current project
cd	Set working directory in IDE
close	Close project in IDE window
dir	Files and directories in current IDE window
display	Properties of IDE handle

<code>getbuildopt</code>	Generate structure of build tools and options
<code>halt</code>	Halt program execution by processor
<code>info</code>	Information about processor
<code>insert</code>	Insert debug point in file
<code>isrunning</code>	Determine whether processor is executing process
<code>isvisible</code>	Determine whether IDE is visible on desktop
<code>listsessions</code>	List existing sessions
<code>load</code>	Load program file onto processor
<code>new</code>	Create project, library, or build configuration in IDE
<code>open</code>	Open project in IDE
<code>profile</code>	Generate real-time execution or stack profiling report
<code>read</code>	Read data from processor memory
<code>remove</code>	Remove file, project, or breakpoint
<code>reset</code>	Stop program execution and reset processor
<code>run</code>	Execute program loaded on processor
<code>save</code>	Save file
<code>setbuildopt</code>	Set active configuration build options
<code>symbol</code>	Program symbol table from IDE
<code>visible</code>	Set whether IDE window is visible while IDE runs
<code>write</code>	Write data to processor memory block
<code>xmakefilesetup</code>	Configure Embedded IDE Link 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 directories in current IDE window
display	Properties of IDE handle
eclipseide	Create handle object to interact with Eclipse IDE
eclipseidesetup	Configure Embedded IDE Link to work 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
pwd	Working directory 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 Embedded IDE Link 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 directory in IDE
close	Close project in IDE window
connect	Connect IDE to processor
dir	Files and directories in current IDE window
display	Properties of IDE handle
getbuildopt	Generate structure of build tools and options
ghsmulti	Create handle object to interact with MULTI IDE
ghsmulticonfig	Configure Embedded IDE Link to work 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
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 Embedded IDE Link to generate makefiles

## Texas Instruments Code Composer Studio

<code>activate</code>	Mark file, project, or build configuration as active
<code>add</code>	Add files to active project in IDE
<code>address</code>	Memory address and page value of symbol in IDE
<code>animate</code>	Run application on processor to breakpoint
<code>build</code>	Build or rebuild current project
<code>ccsboardinfo</code>	Information about boards and simulators known to IDE
<code>cd</code>	Set working directory in IDE
<code>checkEnvSetup</code>	Check system requirements and configure Embedded IDE Link to work with CCS IDE
<code>close</code>	Close project in IDE window
<code>configure</code>	Define size and number of RTDX™ channel buffers
<code>datatypemanager</code>	Open Data Type Manager
<code>dir</code>	Files and directories in current IDE window
<code>disable</code>	Disable RTDX interface, specified channel, or all RTDX channels
<code>display</code>	Properties of IDE handle
<code>enable</code>	Enable RTDX interface, specified channel, or all RTDX channels
<code>flush</code>	Flush data or messages from specified RTDX channels
<code>getbuildopt</code>	Generate structure of build tools and options
<code>halt</code>	Halt program execution by processor

<code>info</code>	Information about processor
<code>insert</code>	Insert debug point in file
<code>isenabled</code>	Determine whether RTDX link is enabled for communications
<code>isreadable</code>	Determine whether MATLAB software can read specified memory block
<code>isrtdxcapable</code>	Determine whether processor supports RTDX
<code>isrunning</code>	Determine whether processor is executing process
<code>isvisible</code>	Determine whether IDE is visible on desktop
<code>iswritable</code>	Determine whether MATLAB software can write to specified memory block
<code>list</code>	Information listings from IDE
<code>load</code>	Load program file onto processor
<code>msgcount</code>	Number of messages in read-enabled channel queue
<code>new</code>	Create project, library, or build configuration in IDE
<code>open</code>	Open project in IDE
<code>profile</code>	Generate real-time execution or stack profiling report
<code>read</code>	Read data from processor memory
<code>readmat</code>	Matrix of data from RTDX channel
<code>readmsg</code>	Read messages from specified RTDX channel
<code>regread</code>	Values from processor registers

<code>regwrite</code>	Write data values to registers on processor
<code>reload</code>	Reload most recent program file to processor signal processor
<code>remove</code>	Remove file, project, or breakpoint
<code>reset</code>	Stop program execution and reset processor
<code>restart</code>	Reload most recent program file to processor signal processor
<code>run</code>	Execute program loaded on processor
<code>save</code>	Save file
<code>setbuildopt</code>	Set active configuration build options
<code>symbol</code>	Program symbol table from IDE
<code>ticcs</code>	Create handle object to interact with CCS IDE
<code>visible</code>	Set whether IDE window is visible while IDE runs
<code>write</code>	Write data to processor memory block
<code>writemsg</code>	Write messages to specified RTDX channel
<code>xmakefilesetup</code>	Configure Embedded IDE Link to generate makefiles



# Functions — Alphabetical List

---

# activate

---

**Purpose** Mark file, project, or build configuration as active

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

**IDEs** This function works with the following IDEs:

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

**Description** Use the `IDE_Obj.activate('objectname','type')` method to make a project file, text 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.

**Inputs** `IDE_Obj`

For `IDE_Obj`, enter the name of the IDE handle object you created using a constructor function. For more information, see “Constructor” on page 8-3.

`objectname`

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

For project and text 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 10-15.

*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.
- 'text' — Gives focus to a specified text file
- 'buildcfg' — Make a specified build configuration active

**IDE support for *type***

	<b>CCS</b>	<b>Eclipse</b>	<b>MULTI</b>	<b>VisualDSP++</b>
'project'	Yes	Yes	Yes	Yes
'text'	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')
```

After opening a file in the IDE window, make the file active:

```
h.activate('text.cpp', 'text')
```

**See Also**

build, new, remove

# add

---

**Purpose** Add files to active project in IDE

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

**IDEs** This function works with the following IDEs:

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

**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 a constructor to create an IDE handle object, such as `IDE_Obj`. For more information, see “Constructor” on page 8-3.
- 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	
DSP/BIOS file (only with CCS IDE and Target Support Package software)	.tcf	DSP/BIOS Config

---

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

---

## Inputs

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, use a constructor function to create *IDE\_Obj*. For more information, see “Constructor” on page 8-3.

*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, Embedded IDE Link searches the IDE working folder first. It then

# add

---

searches the directories 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'.

## Outputs

The add method assigns the type, size, and uclass of the file to *IDE\_Obj.type*.

## 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

**Purpose**

Memory address and page value of symbol in IDE

**Syntax**

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

**IDEs**

This function works with the following IDEs:

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

**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.

**Inputs**

*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 a constructor function to create `IDE_Obj`. For more information, see “Constructor” on page 8-3.

*symbol*

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

# address

---

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.

## Outputs

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 `address` method returns the symbol name, address offset, and page for the symbol as a 1-by-2 vector. The first cell, `a{1}`, contains the symbol name. The second cell contains the address, `a{2}(1)`, and the memory page `a{2}(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  $\pi$ , 12.3,  $e^{-1}$ , and  $\sin(\pi/4)$ . 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`

# adivdsp

---

**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')
```

**IDEs** This function works with 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).

---

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

---

```
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 (refer to “Debug Operations” on page 8-6 for the methods available) 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

- **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 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. **procnum** can also be an array of processor indexes on a multiprocessor board. Using an array results in the adivdsp object **IDE\_Obj** being an array of handles that correspond to the specified processors.

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('sessionname','my_multiproc_session','procnum',[0  
1]);
```

returns a 1-by-2 array of handles to the first and second DSP processor for the multiprocessor session `my_multiproc_session`. `IDE_Obj(1)` is the handle for first processor (0) `IDE_Obj(2)` is the handle for second processor (1).

`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 Embedded IDE Link to work with VisualDSP++ IDE

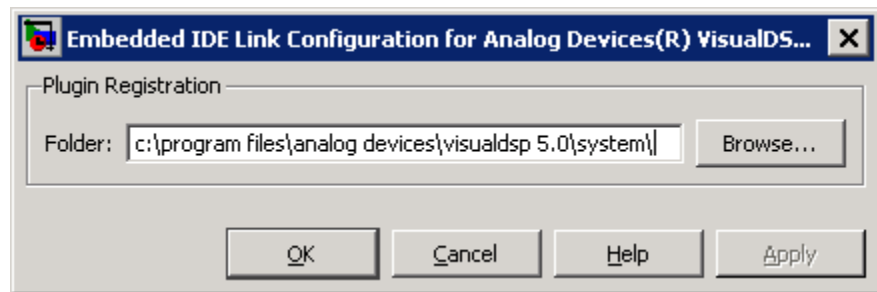
**Syntax**

**IDEs** This function works with the following IDEs:

- Analog Devices VisualDSP++

**Description** Enter `adivdspsetup` at the MATLAB command line when you are setting up Embedded IDE Link to work with VisualDSP++ for the first time. This action displays a dialog box that specifies where Embedded IDE Link installs 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. Also see “Installation and Configuration”.

**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 Embedded IDE Link plugin to the VisualDSP++ IDE.

**See Also** `adivdsp`

**Purpose** Run application on processor to breakpoint

**Syntax** `IDE_Obj.animate`

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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`

# build

---

**Purpose** Build or rebuild current project

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

**IDEs** This function works with the following IDEs:

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

**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 works with the following IDEs:

- Texas Instruments Code Composer Studio

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

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

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 than return the table of the information, the method returns a list of board names and numbers. In that list, each board has a structure named `proc` that contains processor information. For example

```
boards = ccsboardinfo

returns

boards =
```

```

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

```

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

```

boards.proc

ans =

        name: 'CPU'
    number: 0
    type: 'TMS320C6200'

```

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

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

```
boards = ccsboardinfo;
```

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

```

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

```

## Examples

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

```
ccsboardinfo
```

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

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

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

Board Num	Board Name	Proc Num	Processor Name	Processor Type
2	C6xxx Simulator (Texas Instrum .0	0	CPU	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	1	CPU_Primary	TMS320C6200
1	C6xxx EVM (Texas Instrum ...	0	CPU_Secondary	TMS320C6200
0	C64xx Simulator (Texas Instru...0	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

```
ccsboardinfo
```

at the MATLAB desktop prompt. You get

Board Num	Board Name	Proc Num	Processor Name	Processor Type
1	C6xxx Simulator (Texas Instrum .0	0	CPU	TMS320C6211

```

0 C6211 DSK (Texas Instruments) 2 CPU_3 TMS320C6x1x
0 C6211 DSK (Texas Instruments) 1 CPU_4_1 TMS320C6x1x
0 C6211 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=
C6211 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`

---

<b>Purpose</b>	Set working directory in IDE
<b>Syntax</b>	<code>wd=IDE_Obj.cd</code> <code>IDE_Obj.cd(directory)</code>
<b>IDEs</b>	This function works with the following IDEs: <ul style="list-style-type: none"><li>• Analog Devices VisualDSP++</li><li>• Green Hills MULTI</li><li>• Texas Instruments Code Composer Studio</li></ul>
<b>Description</b>	<p><code>wd=IDE_Obj.cd</code> assigns the current working directory of the IDE to the variable, <code>wd</code>, which you reference via the IDE handle object, <code>IDE_Obj</code>.</p> <p><code>IDE_Obj.cd(directory)</code> sets the IDE working directory to 'directory'. 'directory' can be a path string relative to your current working directory, or an absolute path. The intended directory must exist. <code>cd</code> does not create a directory. Setting the IDE directory does not affect your MATLAB working directory.</p> <p><code>cd</code> alters the default directory for <code>open</code> and <code>load</code>. Loading a new workspace file also changes the working directory for the IDE.</p>
<b>See Also</b>	<code>dir</code> <code>load</code> <code>open</code>

# checkEnvSetup

---

**Purpose** Check system requirements and configure Embedded IDE Link to work with CCS IDE

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

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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 work 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 'ccs'.
- 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: 3.3.80.11

Required by : Embedded IDE Link 4.0

Required for : Code generation

2. CGT (Texas Instruments C2000 Code Generation Tools)

Required version: 5.0.2

Required by : Embedded IDE Link 4.0

Required for : Code generation

3. DSP/BIOS (Real Time Operating System)

Required version: 5.32.04

Required by : Embedded IDE Link 4.0

Required for : Code generation

4. Flash Tools (TMS320C2808 Flash APIs)

Required version: 3.02

Required by : Target Support Package 4.0

Required for : Flash Programming

Required environment variables (name, value):

(FLASH\_2808\_API\_INSTALLDIR, "Flash Tools (TMS320C2808 Flash APIs)

# checkEnvSetup

---

After installing or upgrading tools, compare your versions of the tools with the required versions. Use 'check' as the *action* argument:

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

1. Checking CCS (Code Composer Studio) version  
Your Version : 3.3.80.11  
Required version: 3.3.80.11  
Required by : Embedded IDE Link 4.0  
Required for : Code generation
2. Checking DSP/BIOS (Real Time Operating System) version  
Your Version : 5.33.03  
Required version: 5.32.04  
Required by : Embedded IDE Link 4.0  
Required for : Code generation
3. Checking CGT (Code Generation Tools) version  
Your Version : 6.1.5  
Required version: 6.1.4  
Required by : Embedded IDE Link 4.0  
Required for : Code generation

Finally, set the environment variables Embedded IDE Link 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: 3.3.80.11  
Required by : Embedded IDE Link 4.0  
Required for : Code generation  
Your Version : 3.3.80.11
2. Checking DSP/BIOS (Real Time Operating System) version  
Required version: 5.32.04  
Required by : Embedded IDE Link 4.0

```
Required for      : Code generation
Your Version     : 5.33.03
Incompatible version detected. DSP/BIOS version does not satisfy
```

3. Checking CGT (Code Generation Tools) version

```
Required version: 6.1.4
Required by      : Embedded IDE Link 4.0
Required for     : Code generation
Your Version     : 6.1.5
Incompatible version detected. CGT version does not satisfy prod
```

4. Checking DM6437EVM DVSDK (Digital Video Software Developers Kit)

```
Required version: 1.01.00.15
Required by      : Embedded IDE Link 4.0
Required for     : Code generation
Your Version     : 1.01.00.15
### Setting environment variable "DVSDK_EVMDM6437_INSTALLDIR" to
### Setting environment variable "CSLR_DM6437_INSTALLDIR" to "C:\
### Setting environment variable "PSP_EVMDM6437_INSTALLDIR" to "C
### Setting environment variable "NDK_INSTALL_DIR" to "C:\dvsdk_1
```

# close

---

**Purpose** Close project in IDE window

---

**Note** `close( , 'text')` produces an error.

---

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

**IDEs** This function works with the following IDEs:

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

**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** 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

# configure

---

**Purpose** Define size and number of RTDX channel buffers

---

**Note** `configure` produces a warning on C5000 and C6000 processors. We will remove this method from a future version of the software.

---

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

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

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

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

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

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

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

```
TICCS Object:
API version      : 1.0
Processor type   : C67
Processor name   : CPU
Running?        : No
```

```
Board number      : 0
Processor number  : 0
Default timeout   : 10.00 secs
```

```
RTDX channels     : 0
```

```
rx=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

# connect

---

**Purpose** Connect IDE to processor

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

**IDEs** This function works with 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

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

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

**See Also** `load`  
`run`



## Purpose

Open Data Type Manager

`datatypemanager` produces a warning. We will remove this method from a future version of the software.

## Syntax

```
IDE_Obj.datatypemanager  
IDE_Obj2 = IDE_Obj.datatypemanager
```

## IDEs

This function works with the following IDEs:

- Texas Instruments Code Composer Studio

## Description

`IDE_Obj.datatypemanager` opens the Data Type Manager (DTM) with data type information about the project to which `IDE_Obj` refers. With the type manager open, you can add type definitions (typedefs) from your project to MATLAB software so it can interpret them. You add your typedefs because MATLAB software cannot determine or understand typedefs in your function prototypes remotely across the interface to CCS.

Before using the typedef with a function object, the custom type definition in your prototype must appear on the **Typedef name (Equivalent data type)** list.

When the DTM opens, various information and options displays in the Data Type Manager dialog box:

- **Typedef name (Equivalent data type)** — provides a list of default data types. When you create a typedef, it appears added to this list.
- **Add typedef** — opens the **Add Typedef** dialog box so you can add one or more typedefs to your project. Your added typedef appears on the **Typedef name (Equivalent data type)** list. Also, when you pass the `IDE_Obj` object to the DTM, and then add a typedef, the command

```
IDE_Obj.type
```

returns a list of the data types in the object including the typedefs you added.

- **Remove typedef** — removes a selected typedef from the **Typedef name (Equivalent data type)** list.
- **Load session** — loads a previously saved session so you can use the typedefs you defined earlier without reentering them.
- **Refresh list** — updates the list in **Typedef name (Equivalent data type)**. Refreshing the list ensures that the contents are current. Changing your project data type content or loading a new project updates the type definitions in the DTM.
- **Close** — closes the DTM and prompts you to save the session information. This action is the only way to save your work in this dialog box. Saving the session creates a MATLAB file you can reload into the DTM later.

Clicking **Close** in the DTM prompts you to save your session. Saving the session creates a MATLAB file that contains operations that create your final list of data types. These data types are identical to the ones in the **Typedef name** list.

The stored MATLAB file contains a function that includes the add and remove operations you used to create the list of data types in the DTM. For each time you added a typedef in the DTM, the MATLAB file contains an add command that adds the new type definition to the *IDE\_Obj.type* property. When you remove a data type, you see an equivalent clear command that removes a data type from the *IDE\_Obj.type* object.

---

**Note** This method saves to the generated MATLAB file, all the operations that add and remove data types to the DTM during a session. The file includes mistakes you make while creating or removing type definitions. When you load your saved session into the DTM, you see the same error messages you saw during the session. Keep in mind that you have already corrected these errors.

---

The first line of the MATLAB file is a function definition, where the name of the function is the filename of the session you saved.

`IDE_Obj2 = IDE_Obj.datatypemanager` returns the `IDE_Obj2` ticcs object while it opens the DTM. `IDE_Obj2` represents an alias to `IDE_Obj`. Objects `IDE_Obj` and `IDE_Obj2` are not independent objects. When you change a property of either `IDE_Obj` or `IDE_Obj2`, the corresponding property in the other object changes as well.

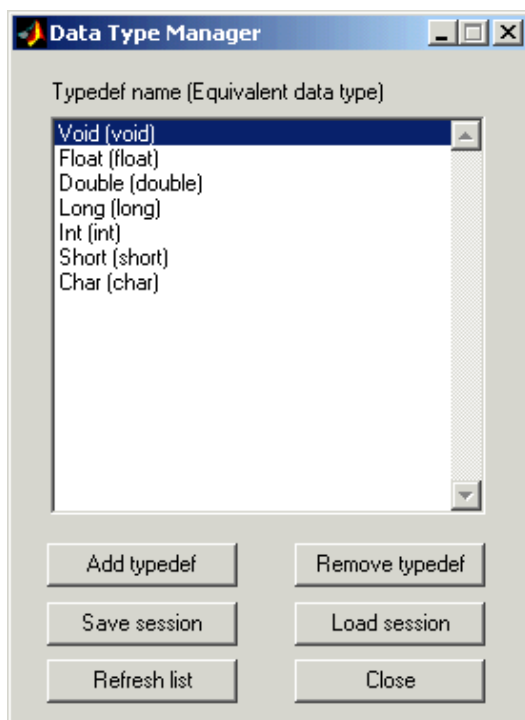
## Data Type Manager

When you create objects that access functions in a project, MATLAB software can recognize most data types that you use in your project. However, if the functions use one or more custom type definitions, MATLAB software cannot recognize the data type and cannot work with the function. To overcome this problem, the Data Type Manager provides the capability to define your typedefs to MATLAB software.

Entering

```
IDE_Obj.datatypemanager
```

at the MATLAB prompt opens the DTM.



Before you add a type definition, the **Typedef name (Equivalent data type)** list shows a number of data types already defined:

- `Void(void)` — void return argument for a function
- `Float(float)` — float data type used in a function input or return argument
- `Double(double)` — double data type used in a function input or return argument
- `Long(long)` — long data type used in a function input or return argument
- `Int(int)` — int data type used in a function input or return argument

- `Short (short)` — short data type used in a function input or return argument
- `Char (char)` — character data type used in a function input or return argument

The lowercase versions of the data types appear because MATLAB software does not recognize the initial capital versions automatically. The data type entry maps the project data type with the initial capital letter to the lowercase MATLAB software data type.

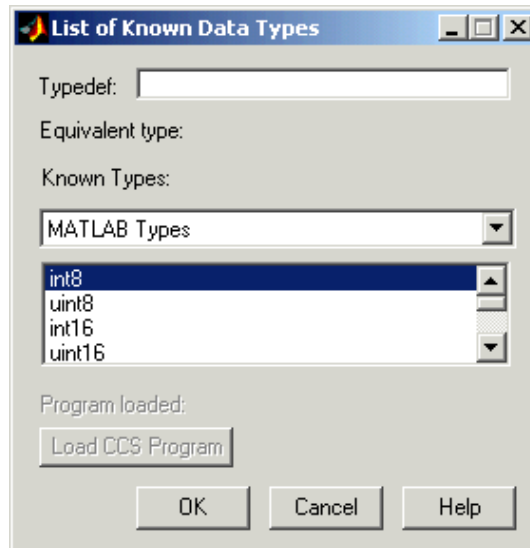
Although not recommended, you can use mixed case typedef names, so long as the equivalent data type uses lowercase. In particular, typedefs that refer to other typedefs are likely to resolve to a data type in lowercase.

Adding a type definition adds the new data type to the list of typedefs.

Remove any existing or new type definitions with the **Remove typedef** option.

## Add Typedef Dialog Box

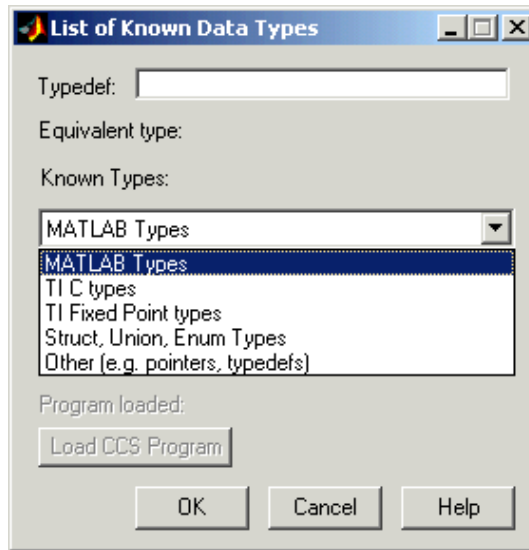
Clicking **Add typedef** in the DTM opens the List of Known Data Types dialog box. As shown in this figure, you add your custom type definitions here.



When you have used custom type definitions in your program or project, specify what they mean to MATLAB software. The **Typedef** option lets you enter the name of the typedef in your program and select an equivalent type from the **Known Types** list. By defining your type definitions in this dialog box, you enable MATLAB software to understand and work with them. For example, when you return the data to the MATLAB workspace or send data from the workspace to your project.

After defining each typedef, the **Equivalent type** option shows the type you specified for each type definition, when you enter it in the **Typedef** field or select it from the **Known Types** list.

Options in this dialog box let you review the data types you are using or that are available in your projects. By selecting different data type categories from the **Known Types** list, you can see all of the supported data types.



From the list of known data types, choose one of the following data type categories:

- MATLAB Types

Data Type	Description
int8	8-bit integer data
uint8	Unsigned 8-bit integer data
int16	16-bit integer data
uint16	Unsigned 16-bit integer data
int32	32-bit integer data

# datatypemanager

---

<b>Data Type</b>	<b>Description</b>
uint32	Unsigned 32-bit integer data
int64	64-bit integer data
uint64	Unsigned 64-bit integer data
single	32-bit IEEE® floating-point data
double	64-bit IEEE floating-point data

- TI C Types

<b>Data Type</b>	<b>Description (For C6000 Compiler)</b>
char	8-bit character data with a sign bit
unsigned char	8-bit character data
signed char	8-bit character data
short	16-bit numeric data
unsigned short	Unsigned 16-bit numeric data
signed short	16-bit numeric data with sign designation
int	32-bit integer numeric data
unsigned int	32-bit integer numerics without sign information
signed int	32-bit integer numerics with sign information
long	40-bit data with sign bit. This type is not the same as int.
unsigned long	40-bit data without information about the sign of the number
signed long	40-bit data without information about the sign of the number represented
float	32-bit numeric data



Data Type	Description (For C6000 Compiler)
double	64-bit numeric data
long double	On the C2000 and C5000 processors – 32-bit IEEE floating-point data On the C6000 processors – 64-bit IEEE floating-point data

Numbers of bits change depending on the processor and compiler. For more information about Texas Instruments data types and specific processors or compilers, refer to your compiler documentation from Texas Instruments processors.

- TI Fixed-Point Types

Data Type	Description
Q0.15	Numeric data with 16-bit word length and 15-bit fraction length
Q0.31	32-bit word length numeric data with fraction length of 31 bits

- Struct, Union, Enum types

If the program you load on the processor includes one or more of `struct`, `union`, or `enum` data types, the type definitions show up on this list. Until you load a program on the processor, this list is empty and trying to access the list generates an error message.

Load a program, if you have not already done so, by clicking **Load CCS Program** and selecting a `.out` file to load on your processor.

- When the load process works, you see the name of the file you loaded in **Loaded program**. Otherwise you get an error message that the load failed.

Only programs that you load from this dialog box appear in **Program loaded**. Programs that you already loaded on your processor do not

appear in the **Loaded program** option. MATLAB software cannot determine what program you have loaded.

- Others such as pointers and typedefs

Like `struct`, `union`, and `enum` data types, the **Others** list is empty until you define one or more typedefs. Unlike the **Struct**, **Union**, **Enum types** list, loading a program does not populate this list with typedefs from the program. Define them explicitly in this dialog box.

Custom type definitions can refer to other typedefs in your project. Nesting typedefs works after you define the necessary custom types. To create a typedef that uses another typedef, define the nested (inner) definition, and then define the outer definition as a pointer to the nested definition. Refer to Examples.

**Program loaded** — if you loaded the program from this dialog box, this parameter tells you the name of the program loaded on the processor. If not, **Program loaded** does not report the program name.

**Load CCS Program** — opens the **Load Program** dialog box so you can select and load a `.out` file to your processor.

## Examples

This set of examples show how to create custom type definitions with the DTM. Each example shows the **List of Known Data Types** dialog box with the selections or entries for creating the typedef.

Start the examples by creating a `ticcs` object:

```
IDE_Obj=ticcs;
```

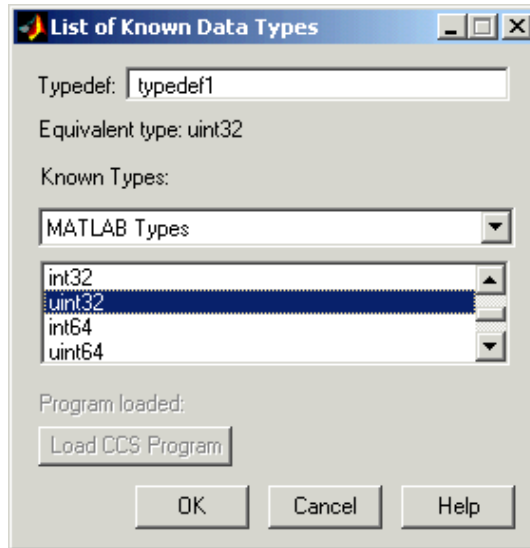
Now start the DTM with the `IDE_Obj` object. So far you have not loaded a file on the processor.

```
IDE_Obj.datatypemanager;
```

With the DTM open, you can create a few custom data types.

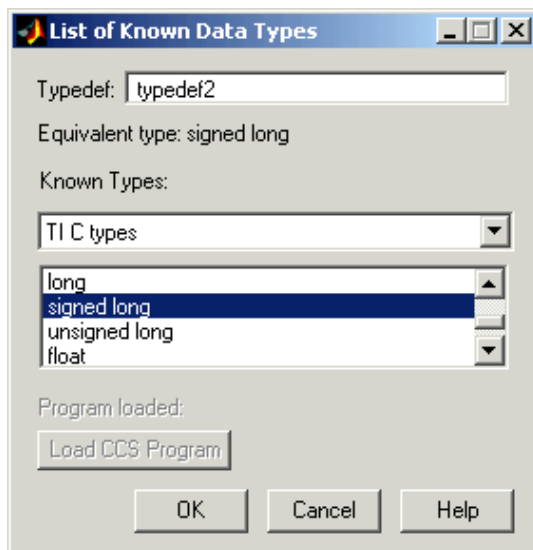
## First Example

Create a typedef (typedef1) that uses a MATLAB software data type. typedef1 uses the equivalent data type uint32.



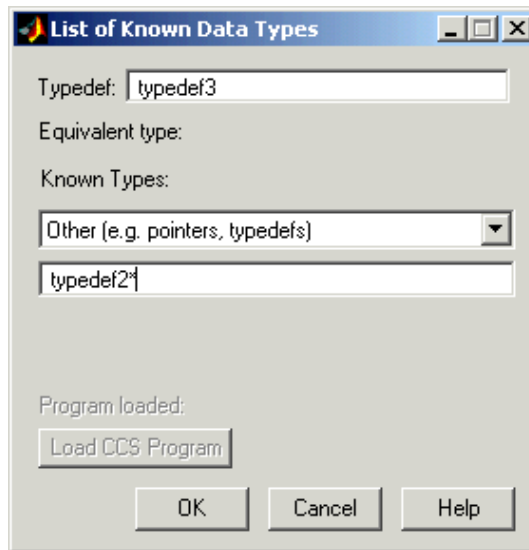
## Second Example

Create a second typedef (typedef2) that uses one of the TI C data types. Define typedef2 to use the signed long data type.



## Third Example

Create a “nested” typedef (typedef3) that refers to another typedef (typedef2).

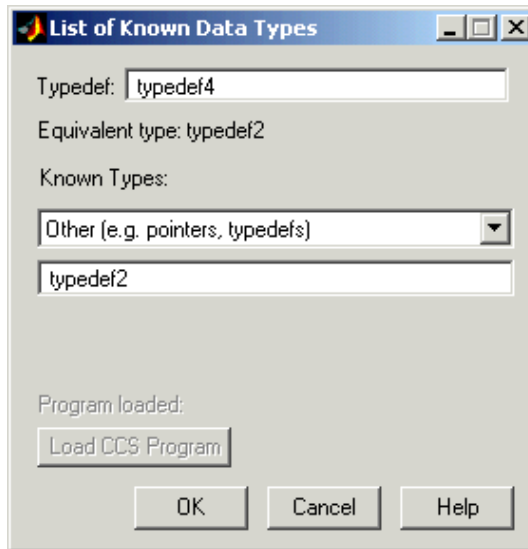


Notice that the referenced typedef, typedef2, is a pointer (indicated by the added asterisk). Using the pointer form lets MATLAB software recognize the data type that typedef2 represents. If you do not use the pointer, MATLAB software converts typedef3 to a default value equivalent data type, in this case, int.

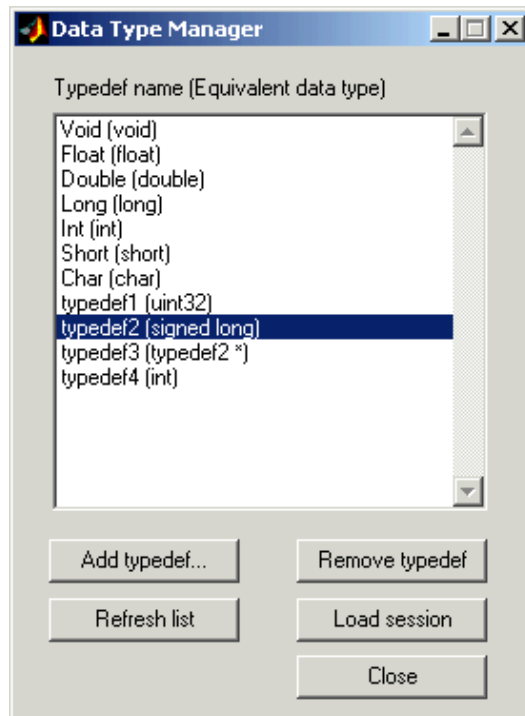
# datatypemanager

---

The next figure shows `typedef4` created to use `typedef2` rather than `typedef2*` for a nested typedef. Under **Equivalent type**, `typedef4` has an equivalent data type of `typedef2`, as specified. But, when you look at the list of known data types in the Data Type Manager dialog box, you see `typedef4` maps to `int`, not `typedef2`, or eventually signed `long`.



Here is the DTM after you create all the example custom data types. Take note of `typedef4` in this listing. You see `typedef4` defaults to an equivalent data type `int`, where `typedef3`, also a nested type definition, retains the equivalent data type you assigned. Now you are ready to use a function that includes your custom type definitions in your hardware-in-the-loop development work.



**Purpose** Files and directories in current IDE window

**Syntax** `IDE_Obj.dir`  
`d=IDE_Obj.dir`

**IDEs** This function works with the following IDEs:

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

**Description** `IDE_Obj.dir` lists the files and directories in the IDE working directory, 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 directories referenced by each object.

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

Field Name	Description
<code>name</code>	Name of the file or directory.
<code>date</code>	Date of most recent file or directory modification.
<code>bytes</code>	Size of the file in bytes. Directories return 0 for the number of bytes.
<code>isdirectory</code>	0 if it is a file, 1 if it is a directory.
<code>datenum</code>	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`

# disable

---

## Purpose

Disable RTDX interface, specified channel, or all RTDX channels

---

**Note** Support for `disable` on C5000 and C6000 processors will be removed in a future version.

---

## Syntax

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

## IDEs

This function works with the following IDEs:

- Texas Instruments Code Composer Studio

## 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

# display

---

**Purpose** Properties of IDE handle

**Syntax** `IDE_Obj.display()`

**IDEs** This function works with the following IDEs:

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

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

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

```
IDE_Obj.display
```

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

**See Also** `get` in the MATLAB Function Reference

**Purpose** Create handle object to interact with Eclipse IDE

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

**IDEs** This function works with 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 “Getting Started” topic for Eclipse IDE.
- Use the `eclipseidesetup` function to configure and install a plug-in that enables Embedded IDE Link product to work 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 listed in “Eclipse IDE” on page 8-11.

When you use `eclipseide`, the Embedded IDE Link software uses the plug-in to open a session with Eclipse. If Eclipse 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, Embedded IDE Link 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: `IDE_Obj = eclipseide('timeout', period)`

For *period*, enter the number of seconds the handle object waits for processor operations (such as load) to complete. Operations that exceed

# eclipseide

---

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

**Purpose**

Configure Embedded IDE Link to work with Eclipse IDE

**Syntax****IDEs**

This function works with 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 “Getting Started” 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 Embedded IDE Link to work with Eclipse IDE. This action displays a dialog box which you use to configure and add an Embedded IDE Link plugin to the Eclipse IDE. For detailed instructions and examples, see “[Configuring Embedded IDE Link 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 and C6000 processors will be removed in a future version.

---

## Syntax

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

## IDEs

This function works with the following IDEs:

- Texas Instruments Code Composer Studio

## 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

# flush

---

## Purpose

Flush data or messages from specified RTDX channels

---

**Note** `flush` support for C5000 and C6000 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 works with the following IDEs:

- Texas Instruments Code Composer Studio

## 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, Embedded IDE Link 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`

# getbuilddopt

---

**Purpose** Generate structure of build tools and options

**Syntax** `bt=IDE_Obj.getbuilddopt`  
`cs=IDE_Obj.getbuilddopt(file)`

**IDEs** This function works with the following IDEs:

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

**Description** `bt=IDE_Obj.getbuilddopt` 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.getbuilddopt(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**

Create handle object to interact with MULTI IDE

**Syntax**

```
IDE_Obj = ghsmulti
IDE_Obj=ghsmulti('propertyname1',propertyvalue1,'propertyname2',...
propertyvalue2,'timeout',value)
```

**IDEs**

This function works with 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 Embedded IDE Link service.

ghsmulti creates an interface between MATLAB and Green Hills MULTI. If this is the first time you have used 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 Embedded IDE Link service. The default host name indicates that the service is on the local PC. Replace localhost with the name you entered in <b>Host name</b> on the Embedded IDE Link Configuration dialog box.
portnum	4444	Specifies the port to connect to the Embedded IDE Link service on the host machine. Replace portnum with the number

Property Name	Default Value	Description
		you entered in <b>Port number</b> on the Embedded IDE Link Configuration dialog box.

When you invoke `ghsmulti`, it starts the Embedded IDE Link service. If you selected the **Show server status window** option on the Embedded IDE Link Configuration dialog box (refer to `ghsmulticonfig`) when you configured your MULTI installation, 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).

---

**Note** The output object name you provide for `ghsmulti` cannot begin with an underscore, such as `_IDE_Obj`.

---

`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, Embedded IDE Link 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 (refer to “Debug Operations” on page 8-6 for the methods available) 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 Embedded IDE Link to work with MULTI IDE

**Syntax** `ghsmulticonfig`

**IDEs** This function works with the following IDEs:

- Green Hills MULTI

**Description** `ghsmulticonfig` launches the Embedded IDE Link Configuration dialog box that you use to configure your Embedded IDE Link software installation to work with MULTI.

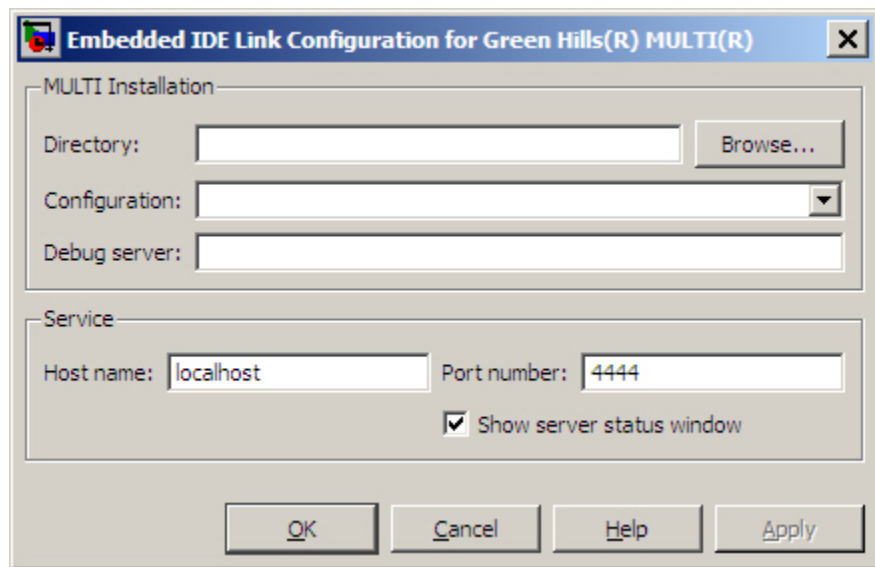
---

**Note** The Embedded IDE Link 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

Tells Embedded IDE Link software the path to your Green Hills MULTI software installation. Enter the full path to the Green Hills MULTI executable, `multi.exe`, in your installation. To search for the executable file, click **Browse**.

If you do not provide or select a correct path to the executable file, Embedded IDE Link software ignores your entry and returns an error message saying it could not find the executable `multi.exe` in the specified or selected directory.

### 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`.

Embedded IDE Link 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. Embedded IDE Link 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 ffffffff
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 Embedded IDE Link service. Enter `localhost` if the service runs on your PC. `localhost` is the only supported host name.

**Port number**

Specify the port the Embedded IDE Link 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, Embedded IDE Link software returns an error when you try to create a `ghsmulti` object.

**Show server status window**

Select this option to display the Embedded IDE Link 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.

# halt

---

**Purpose** Halt program execution by processor

**Syntax** `IDE_Obj.halt`  
`IDE_Obj.halt(timeout)`

**IDEs** This function works with the following IDEs:

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

**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 works with the following IDEs:

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

**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:

<b>Structure Element</b>	<b>Data Type</b>	<b>Description</b>
<code>adf.CurBrkPt</code>	String	When the debugger is stopped at a breakpoint, the field reports the index of the breakpoint. Otherwise, this value is -1.
<code>adf.File</code>	String	Name of the current file shown in the debugger source pane.
<code>adf.Line</code>	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
<code>adf.MultiDir</code>	String	Full path to your IDE installation the root directory). For example  <code>'C:\ghs5_01'</code>
<code>adf.PID</code>	Double	Process ID from the debug server in the IDE.
<code>adf.Procedure</code>	String	Current procedure in the debugger source pane.
<code>adf.Process</code>	Double	Program number, defined by the IDE, of the current program.
<code>adf.Remote</code>	String	Status of the remote connection, either <code>Connected</code> or <code>Not connected</code> .
<code>adf.Selection</code>	String	The string highlighted in the debugger. If there is no string highlighted, this value is <code>'null'</code> .

# info

Structure Element	Data Type	Description
adf.State	String	State of the loaded program. The possible reported states appear in the following list: <ul style="list-style-type: none"><li>• About to resume</li><li>• Dying</li><li>• Just executed</li><li>• Just forked</li><li>• No child</li><li>• Running</li><li>• Stopped</li><li>• Zombied</li></ul> 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
<code>adf.procname</code>	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with <code>IDE_Obj</code> .
<code>adf.isbigendian</code>	Boolean	Value describing the byte ordering used by the processor. When the processor is big-endian, this value is 1. Little-endian processors return 0.
<code>adf.family</code>	Integer	Three-digit integer that identifies the processor family, ranging from 000 to 999. For example, 320 for Texas Instruments digital signal processors.
<code>adf.subfamily</code>	Decimal	Decimal representation of the hexadecimal identification value that TI assigns to the processor to identify the processor subfamily. IDs range from 0x000 to 0x3822. Use <code>dec2hex</code> to convert the value in <code>adf.subfamily</code> to standard notation. For example  <code>dec2hex(adf.subfamily)</code>  produces '67' when the processor is a member of the 67xx processor family.
<code>adf.timeout</code>	Integer	Default timeout value MATLAB software uses when transferring data to and from CCS. All functions that use a timeout value have an optional <code>timeout</code> input argument. When you omit the optional argument, MATLAB software uses this default value – 10s.

`adf = info(rx)` returns `info` as a cell array containing the names of your open RTDX channels.

### Examples

On a PC with a simulator configured in CCS IDE, `info` returns the configuration for the processor being simulated:

```
IDE_Obj.info

ans =

    procname: 'CPU'
  isbigendian: 0
      family: 320
  subfamily: 103
    timeout: 10
```

This example simulates the TMS320C6211 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
<code>adf.procname</code>	String	Processor name as defined in the CCS setup utility. In multiprocessor systems, this name reflects the specific processor associated with <code>IDE_Obj</code> .
<code>adf.proctype</code>	String	String with the type of the DSP processor. The type property is the processor type like "ADSP-21065L" or "ADSP-2181".
<code>adf.revision</code>	String	String with the silicon revision string of the processor.

`adf = info(rx)` returns `info` as a cell array containing the names of your open RTDX channels.

### Examples

When you have an `adivdsp` object `IDE_Obj`, `info` provides information about the object:

```
IDE_Obj = adivdsp('sessionname','Testsession')
```

```
ADIVDSP Object:  
Session name      : Testsession  
Processor name    : ADSP-BF533  
Processor type    : ADSP-BF533  
Processor number  : 0  
Default timeout   : 10.00 secs
```

```
objinfo = IDE_Obj.info

objinfo =

    procname: 'ADSP-BF533'
    proctype: 'ADSP-BF533'
    revision: ''

objinfo.procname

ans =

ADSP-BF533
```

**See Also**

dec2hex, get, set

# insert

---

**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 works with the following IDEs:

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

**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 above, 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 above, 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 and C6000 processors will be removed in a future version.

---

**Syntax** `isenabled(rx, 'channel')`  
`isenabled(rx)`

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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` to work, as well as for `enabled` to work. 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 MATLAB software can read specified memory block

---

**Note** Support for `isreadable(rx, 'channel')` on C5000 and C6000 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 works with the following IDEs:

- Texas Instruments Code Composer Studio

**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 <i>IDE_Obj.page</i>
10	Decimal	Address is 10 decimal on the page referred to by <i>IDE_Obj.page</i>
[18,1]	Vector	Address location 10 decimal on memory page 1 ( <i>IDE_Obj.page</i> = 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`. Note that 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:

# isreadable

<b><i>datatype</i> String</b>	<b>Number of Bytes/Value</b>	<b>Description</b>
'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 processor family and  $2^{16}$  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 processor family and  $2^{16}$  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
```

# isreadable

---

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 and C6000 processors will be removed in a future version.

---

**Syntax** `b=IDE_Obj.isrtdxcapable`

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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. It should.

```
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 works with the following IDEs:

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

**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

# isvisible

---

**Purpose** Determine whether IDE is visible on desktop

**Syntax** `IDE_Obj.isvisible`

**IDEs** This function works with the following IDEs:

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

**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, do one of the following.

- Open MATLAB software. Create a link to the IDE. Use the new link to make the IDE visible. Close the IDE.

- Open Microsoft Windows Task Manager. Click **Processes**. Find and highlight IDE\_Obj\_app.exe. Click **End Task**.

## See Also

info, visible

# iswritable

---

**Purpose** Determine whether MATLAB software can write to specified memory block

---

**Note** Support for `iswritable(rx, 'channel')` on C5000 and C6000 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 works with the following IDEs:

- Texas Instruments Code Composer Studio

**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 <code>IDE_Obj.page</code>
10	Decimal	Address is 10 decimal on the page referred to by <code>IDE_Obj.page</code>
[18,1]	Vector	Address location 10 decimal on memory page 1 ( <code>IDE_Obj.page = 1</code> )

To specify the address in hexadecimal format, enter the address property value as a string. `iswritable` interprets the string as the hexadecimal representation of the desired memory location. To convert the hex value to a decimal value, the function uses `hex2dec`. Note that 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]

# iswritable

---

```
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:

<b>datatype String</b>	<b>Description</b>
'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 processor family and  $2^{16}$  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=
```

# iswritable

---

0

Now that your script knows that it can write to 'ichanne'1, it proceeds to write messages as required.

## See Also

hex2dec, isreadable, read



**Purpose**

Information listings from IDE

**Syntax**

```
IDE_Obj.infolist = list('type')
IDE_Obj.infolist = list('type',typename)
list(ff,varname)
infolist = IDE_Obj.list('type')
infolist = IDE_Obj.list('type',typename)
```

**IDEs**

This function works with the following IDEs:

- Green Hills MULTI
- Texas Instruments Code Composer Studio

**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.

# list

---

<b>type String</b>	<b>Description</b>
<b>project</b>	Return information about the current project in the IDE
<b>variable</b>	Return information about one or more embedded variables
<b>function</b>	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.

Also, `list` may report incorrect information when you make changes to variables from MATLAB. To report variable information, `list` uses the IDE API, which only knows about variables in the IDE. Your changes from MATLAB, such as changing the data type of a variable, do not appear through the API and `list`. For example, the following operations return incorrect or old data information from `list`.

`infolist = IDE_Obj.list('project')` returns a vector of structures that contain project information in the format shown in the following table.

<b>infolist Structure Element</b>	<b>Description</b>
<code>infolist(1).name</code>	Project file name (with path)
<code>infolist(1).primary</code>	Configuration file used for the project. For more information, refer to <code>new</code>
<code>infolist(1).compileroptions</code>	Compiler options string for the project

infolist Structure Element	Description
<code>infolist(1).srcfiles</code>	Vector of structures that describes project source files. Each structure contains the name and path for each source file— <code>infolist(1).srcfiles.name</code>
<code>infolist(1).type</code>	Shows the project type, either <code>project</code> or <code>projlib</code> . For more information, refer to <code>new</code> .
<code>infolist(2)....</code>	...
<code>infolist(n)....</code>	...

`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
<code>infolist.varname(1).name</code>	Symbol name.
<code>infolist.varname(1).isglobal</code>	Indicates whether symbol is global or local.
<code>infolist.varname(1).location</code>	Information about the location of the symbol.
<code>infolist.varname(1).size</code>	Size per dimension.

# list

---

infolist Structure Element	Description
<code>infolist.varname(1).uclass</code>	IDE handle class that matches the type of this symbol.
<code>infolist.varname(1).bitsize</code>	Size in bits. More information is added to the structure depending on the symbol type.
<code>infolist.(varname1).type</code>	Data type of symbol.
<code>infolist.varname(2)....</code>	...
<code>infolist.varname(n)....</code>	...

`list` uses the variable name as the field name to refer to the structure information for the variable.

`infolist = 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 format below when you specify option type as **function**:

<b>infolist Structure Element</b>	<b>Description</b>
<code>infolist.functionname(1).name</code>	Function name
<code>infolist.functionname(1).filename</code>	Name of file where function is defined
<code>infolist.functionname(1).address</code>	Relevant address information such as start address and end address
<code>infolist.functionname(1).funcvar</code>	Variables local to the function
<code>infolist.functionname(1).uclass</code>	IDE handle class that matches the type of this symbol— <b>function</b>
<code>infolist.functionname(1).funcdecl</code>	Function declaration—where information such as the function return type is contained
<code>infolist.functionname(1).islibfunc</code>	Is this a library function?
<code>infolist.functionname(1).linepos</code>	Start and end line positions of function
<code>infolist.functionname(1).funcinfo</code>	Miscellaneous information about the function
<code>infolist.functionname(2)...</code>	...
<code>infolist.functionname(n)...</code>	...

To refer to the function structure information, `list` uses the function name as the field name.

# list

---

`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 format below when you specify option type as **type**:

infolist Structure Element	Description
<code>infolist.typename(1).type</code>	Type name
<code>infolist.typename(1).size</code>	Size of this type
<code>infolist.typename(1).uclass</code>	IDE handle class that matches the type of this symbol. Additional information is added depending on the type
<code>infolist.typename(2)...</code>	...
<code>infolist.typename(n)...</code>	...

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

`list(ff,varname)` lists the local variables associated with the function accessed by function object `ff`. Compare to `IDE_Obj.list('variable','varname')` which works the same way to return variables from ticcs object `IDE_Obj`.

---

**Note** `list` does not recognize or return information about variables that you declare in your code but that are not used or initialized.

---

Some restrictions apply when you use `list` with function objects. `list` generates an error in the following circumstances:



- When `varname` is not a valid input argument for the function accessed by `ff`

For example, if your function declaration is

```
int foo(int a)
```

but you request information about input argument `b`, which is not defined

```
list(ff, 'b')
```

MATLAB software returns an error.

- When `varname` is the same as a variable assigned by MATLAB software. Usually this happens when you use `declare` to pass a function declaration to MATLAB software and the declaration string does not match the declaration for `ff` as determined when you created `ff`.

In an example that demonstrates this problem, the function declaration has a name for the first input, `a`. In the `declare` call, the declaration string does not provide a name for the first input, just a data type, `int`. When you issue the `declare` call, MATLAB software names the first input `ML_Input1`. If you try to use `list` to get information about the input named `ML_Input`, `list` returns an error. Here is the code, starting with the function declaration in your code:

```
int foo(int a) % Function declaration in your source code
declare(ff, 'decl', 'int foo(int)')
% MATLAB generates a warning that it has assigned the name
% ML_Input to the first input argument
list(ff, 'ML_Input') % list returns an error for this call
```

- When `varname` does not match the input name in the function declaration provided in your source code, as compared to the declaration string you used in a `declare` operation.

Assume your source code includes a function declaration for `foo`:

```
int foo(int a);
```

Now pass a declaration for `foo` to MATLAB software:

```
declare(ff,'decl','int foo(int b)')
```

MATLAB software issues a warning that the input names do not match. When you use `list` on the input argument `b`,

```
list(ff,'b')
```

`list` returns an error.

- When `varname` is an input to a library function. `list` always fails in this case. It does not matter whether you use `declare` to provide the declaration string for the library function.

---

**Note** When you call `list` for a variable in a function object `list(ff,varname)` the `address` field may contain an incorrect address if the program counter is not within the scope of the function that includes `varname` when you call `list`.

---

`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.
- **type** — Tell `list` to return information about one or more defined data types, including `struct`, `enum`, and `union`. ANSI C data type typedefs are excluded from the list of data types.

Note, 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 `info` 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, such as changing the data type of a variable, 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  
type: 'short *'  
bitsize: 16  
reftype: 'short'  
referent: [1x1 struct]
```

```
member_pts_to_same_struct: 0
name: 'signedShortArray1'
```

The `type` field reports the original data type `short`.

You get this is 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.

`infolist = IDE_Obj.list('project')` returns a vector of structures containing project information in the format shown here when you specify option type as **project**.

<b>infolist Structure Element</b>	<b>Description</b>
<code>infolist(1).name</code>	Project file name (with path).
<code>infolist(1).type</code>	Project type — <code>project</code> , <code>projlib</code> , or <code>projext</code> , refer to <code>new</code>
<code>infolist(1).procesortype</code>	String description of processor CPU
<code>infolist(1).srcfiles</code>	Vector of structures that describes project source files. Each structure contains the name and path for each source file — <code>infolist(1).srcfiles.name</code>
<code>infolist(1).buildcfg</code>	Vector of structures that describe build configurations, each with the following entries: <ul style="list-style-type: none"><li>• <code>infolist(1).buildcfg.name</code> — the build configuration name</li><li>• <code>infolist(1).buildcfg.outpath</code> — the default directory for storing the build output.</li></ul>

infolist Structure Element	Description
<code>infolist(2)....</code>	...
<code>infolist(n)....</code>	...

`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. Note, 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 format below when you specify option `type` as **variable**:

infolist Structure Element	Description
<code>infolist.varname(1).name</code>	Symbol name
<code>infolist.varname(1).isglobal</code>	Indicates whether symbol is global or local
<code>infolist.varname(1).location</code>	Information about the location of the symbol
<code>infolist.varname(1).size</code>	Size per dimension
<code>infolist.varname(1).uclass</code>	ticcs object class that matches the type of this symbol
<code>infolist.varname(1).bitsize</code>	Size in bits. More information is added to the structure depending on the symbol type.
<code>infolist.(varname1).type</code>	data type of symbol
<code>infolist.varname(2)....</code>	...
<code>infolist.varname(n)....</code>	...

# list

---

`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 format below when you specify option type as **function**:

<b>infolist Structure Element</b>	<b>Description</b>
<code>infolist.functionname(1).name</code>	Function name
<code>infolist.functionname(1).filename</code>	Name of file where function is defined
<code>infolist.functionname(1).address</code>	Relevant address information such as start address and end address
<code>infolist.functionname(1).funcvar</code>	Variables local to the function

<b>infolist Structure Element</b>	<b>Description</b>
<code>infolist.functionname(1).uclass</code>	ticcs object class that matches the type of this symbol — <b>function</b>
<code>infolist.functionname(1).funcdecl</code>	Function declaration — where information such as the function return type is contained
<code>infolist.functionname(1).islibfunc</code>	Is this a library function?
<code>infolist.functionname(1).linepos</code>	Start and end line positions of function
<code>infolist.functionname(1).funcinfo</code>	Miscellaneous information about the function
<code>infolist.functionname(2)...</code>	...
<code>infolist.functionname(n)...</code>	...

To refer to the function structure information, `list` uses the function name as the field name.

`infolist = IDE_Obj.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 ANSI C struct tag, enum tag or union tag. If the ANSI C tag is not defined, it is referred to by the CCS compiler as '\$faken' where *n* is an assigned number.

`infolist = IDE_Obj.list('type', typename)` returns a structure that contains information on the specified defined data type.

`infolist = IDE_Obj.list('type', typenamelist)` returns a structure that contains information on the specified defined data types in the list.

The returned information follows the format below when you specify option type as **type**:

<b>infolist Structure Element</b>	<b>Description</b>
<code>infolist.typeName(1).type</code>	Type name
<code>infolist.typeName(1).size</code>	Size of this type
<code>infolist.typeName(1).uclass</code>	ticcs object class that matches the type of this symbol. Additional information is added depending on the type
<code>infolist.typeName(2)...</code>	...
<code>infolist.typeName(n)...</code>	...

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 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

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 CCS, you see a listing like the following that includes structures containing details about your project.

# list

---

```
projectinfo=IDE_Obj.list('project')

projectinfo =

    name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'
    type: 'project'
    processor: 'TMS320C67XX'
    srcfiles: [69x1 struct]
    buildcfg: [3x1 struct]
```

## See Also

[info](#)

<b>Purpose</b>	List existing sessions
<b>Syntax</b>	<pre>list = listsessions list = listsessions('verbose')</pre>
<b>IDEs</b>	This function works with the following IDEs: <ul style="list-style-type: none"><li>• Analog Devices VisualDSP++</li></ul>
<b>Description</b>	<p><code>list = listsessions</code> returns <code>list</code> that contains a listing of all of the sessions by name currently in the development environment.</p> <p><code>list = listsessions('verbose')</code> adds the optional input argument <code>verbose</code>. When you include the <code>verbose</code> argument, <code>listsessions</code> returns a cell array that contains one row for each existing session. Each row has three columns — processor type, platform name, and processor name.</p>
<b>See Also</b>	<code>adivdsp</code>

# load

---

**Purpose** Load program file onto processor

**Syntax** `IDE_Obj.load(filename,timeout)`

**IDEs** This function works with the following IDEs:

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

**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 current working directory of the IDE.

With the VisualDSP++, MULTI, and Code Composer Studio IDEs, you can use the `cd` method to check or modify the IDE working directory.

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
```

# msgcount

---

**Purpose** Number of messages in read-enabled channel queue

---

**Note** Support for msgcount on C5000 and C6000 processors will be removed in a future version.

---

**Syntax** msgcount(rx, 'channel')

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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 works with the following IDEs:

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

**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 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 directory.

If you omit the `type` argument, and the `name` argument does not include a file extension, `type` defaults to 'project'.

## new

---

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'  
IDE_Obj.new('my_build_config','buildcfg') #Create a build configuration
```

### See Also

activate  
close



**Purpose**

Open project in IDE

---

**Note** `open( , 'text' )` produces an error.

`open( , 'program' )` produces an error. Use `load` instead.

`open( , 'workspace' )` produces an error.

---

**Syntax**

`IDE_Obj.open(filename, filetype, timeout)`  
`IDE_Obj.open(myproject)`

**IDEs**

This function works with the following IDEs:

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

**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 directory path, MATLAB returns a warning and returns control to MATLAB.

For the optional *filetype* argument, you can specify the following types:

# open

---

	CCS IDE	Eclipse IDE	MULTI IDE	VisualDSP++ IDE
'project' — Project files	Yes	Yes	Yes	Yes
'ProjectGroup' — Project group files				Yes
'program' — Target program file (executable)	No. Use load instead.		Yes	

If you omit the *filetype* argument, *filetype* defaults to 'project'. The 'text' and 'workspace' options are no longer supported.

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. Usually the program load process works correctly in spite of the error message.

## Examples

`IDE_Obj.open(myproject)` opens the myproject project in the IDE.

## See Also

`cd`  
`dir`  
`load`  
`new`

**Purpose** Generate real-time execution or stack profiling report

**Syntax** `IDE_Obj.profile(type,action,timeout)`

**IDEs** This function works with the following IDEs:

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

**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, using Embedded IDE Link with the Eclipse IDE supports execution profiling for ARM processors running Linux, as follows:

	Windows Platform	Linux Platform
Intel x86/Pentium	No	No

# profile

---

AMD K5/K6/Athlon	No	No
ARM	No	Yes

To get a real-time task execution profile report in HTML and graphical plot forms, set the *type* argument to 'execution' and omit the *action* argument, which defaults to 'report'. For more information, see “Profiling Code Execution in Real-Time” on page 5-10.

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 “System Stack Profiling” on page 5-18.

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.

---

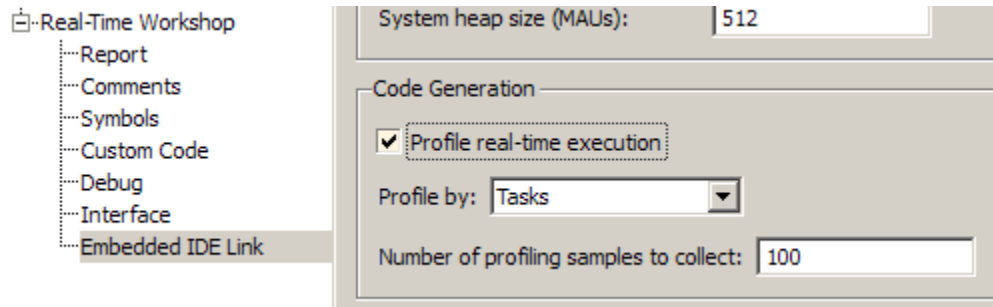
**Note** Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

---

## 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**) and enable **Profile real-time execution**.



- 2 Build your model.

```
IDE_Obj.build
```

- 3 Load your program to the processor.

```
IDE_Obj.load('c:\work\sumdiff.out')
```

- 4 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.

- 5 Run the program on the processor.

```
IDE_Obj.run
```

- 6 Stop the running program.

```
IDE_Obj.halt
```

- 7 To get the profiling reports enter one of the following commands:

```
IDE_Obj.profile('stack','report') #Get the stack profiling report
IDE_Obj.profile('execution') #Get the execution profiling report
```

The HTML report contains the sections described in the following table.

Section Heading	Description
Worst case task turnaround times	Maximum task turnaround time for each task since model execution started.
Maximum number of concurrent overruns for each task	Maximum number of concurrent task overruns since model execution started.
Analysis of profiling data recorded over <i>nnn</i> seconds.	Profiling data was recorded over <i>nnn</i> seconds. The recorded data for task turnaround times and task execution times is presented in the table below 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

# pwd

---

<b>Purpose</b>	Working directory used by Eclipse
<b>Syntax</b>	<code>wd= IDE_Obj .pwd</code>
<b>IDEs</b>	This function works with the following IDEs: <ul style="list-style-type: none"><li>• Eclipse IDE</li></ul>
<b>Description</b>	Use <code>wd= IDE_Obj .pwd</code> to get the working directory of the Eclipse IDE. This value is the same as the Eclipse IDE workspace directory.
<b>Examples</b>	To get the Eclipse IDE working directory: <pre>IDE_Obj = eclipseide; wd = IDE_Obj.pwd  wd =  C:\WINNT\Profiles\rdlugyhe\workspace</pre>
<b>See Also</b>	<code>dir</code>



**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 works with the following IDEs:

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

**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 (see below).

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:

<b>address Parameter Value</b>	<b>Description</b>
131082	Decimal address specification. The memory start address is 131082 and memory type is 0. This is the same as specifying [131082 0].
[131082 1]	Decimal address specification. The memory start address is 131082 and memory type is 1.
'2000A'	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify `address` as a cell array. You can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1}=131072;
myaddress1{2}='Program(PM) Memory';
```

```
myaddress2 myaddress2{1}='20000';
myaddress2{2}='Program(PM) Memory';
```

```
myaddress3 myaddress3{1}=131072; myaddress3{2}=0;
```

`mem=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:

<b>MATLAB Data Type</b>	<b>Description</b>
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

`read` 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_Obj.memorytype` 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`

# readmat

---

## Purpose

Matrix of data from RTDX channel

---

**Note** Support for readmat on C5000 and C6000 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 works with the following IDEs:

- Texas Instruments Code Composer Studio

## 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');
```

# readmat

---

```
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



**Purpose** Read messages from specified RTDX channel

---

**Note** Support for readmsg on C5000 and C6000 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 works with the following IDEs:

- Texas Instruments Code Composer Studio

**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.

# readmsg

---

<b>datatype String</b>	<b>Specified Data Type</b>
'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. `readmsg`s 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 = [1,1]`, 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)
```

# readmsg

---

```
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`

**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 works with the following IDEs:

- Green Hills MULTI
- Texas Instruments Code Composer Studio

**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

# regread

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 <code>represent</code> argument.
'binary'	Source register contains an unsigned binary integer.
'ieee'	Source register contains a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are reading from 32 and 64 bit registers on the target.

To limit the time that `regread` spends transferring data from the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the read operation. You might find this

useful for limiting prolonged data transfer operations. If you omit the *timeout* argument, `regread` defaults to the global time-out defined in `IDE_Obj`.

`reg = 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 during intermediate times in program operation may not get reflected in the register.

This is true for local variables as well.

One way to see this is to write a line of code that uses the variable and see if the result is consistent.

```
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 Embedded IDE Link software .

## Example

### 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`

# regwrite

---

## 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 works with the following IDEs:

- Green Hills MULTI
- Texas Instruments Code Composer Studio

## 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 <i>represent</i> 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` (called a *direct memory write*) to write memory-mapped registers.

---

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, ISTR, 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 during intermediate times in program operation may not get reflected in the register.

This is true for any local variables as well.

One way to see this is to write a line of code that uses the variable and see if result is consistent.

```
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 Embedded IDE Link 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`

# reload

---

**Purpose** Reload most recent program file to processor signal processor

**Syntax**  
`s = IDE_Obj.reload(timeout)`  
`s = IDE_Obj.reload`

**IDEs** This function works with the following IDEs:

- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio

**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 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 =

''

openIDE_Obj('D:\ti\tutorial\sim62xx\gelsolid\hellodsp.pjt',... #This path varies by IDE
'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`

# remove

---

**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 works with the following IDEs:

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

**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.

**Inputs** *IDE\_Obj*

Enter the name of the IDE link handle for your IDE. Use a constructor to create an IDE link handle before you use the remove method. See “Constructor” on page 8-3.

*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.

## remove

---

### **See Also**

add

cd

open

---

<b>Purpose</b>	Stop program execution and reset processor
<b>Syntax</b>	<code>IDE_Obj.reset(timeout)</code>
<b>IDEs</b>	This function works with the following IDEs: <ul style="list-style-type: none"><li>• Analog Devices VisualDSP++</li><li>• Green Hills MULTI</li><li>• Texas Instruments Code Composer Studio</li></ul>
<b>Description</b>	<p><code>IDE_Obj.reset(timeout)</code> stops the program executing on the processor and asynchronously performs a processor reset, returning all processor register contents to their power-up settings. <code>reset</code> returns immediately after the processor halt.</p> <p>The optional <code>timeout</code> argument sets the number of seconds MATLAB waits for the processor to halt. If you omit the <code>timeout</code> argument, <code>timeout</code> defaults to the <code>timeout</code> value of the IDE handle object.</p>
<b>See Also</b>	<code>halt</code> <code>load</code> <code>run</code>

# restart

---

**Purpose** Reload most recent program file to processor signal processor

**Syntax** `IDE_Obj.restart`  
`IDE_Obj.restart(timeout)`

**IDEs** This function works with the following IDEs:

- Eclipse IDE
- Green Hills MULTI
- Texas Instruments Code Composer Studio

**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`

**Purpose**

Execute program loaded on processor

**Syntax**

```
IDE_Obj.run  
IDE_Obj.run('runopt')  
IDE_Obj.run(...,timeout)
```

**IDEs**

This function works with the following IDEs:

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

**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 than one processor, each processor 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.

<b>runopt string</b>	<b>Description</b>
'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'	<p>This option must be followed by an extra parameter <i>funcname</i>, the name of the function to run to:</p> <pre>IDE_Obj.run('tofunc', funcname)</pre> <p>This executes a run from the present PC location until the start of function <i>funcname</i> is reached. If <i>funcname</i> is not along the program's normal execution path, <i>funcname</i> is not reached and the method times out.</p>

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

# save

---

**Purpose** Save file

---

**Note** `IDE_Obj.save( , 'text' )` produces an error.

---

**Syntax** `IDE_Obj.save(filename, filetype)`

**IDEs** This function works with the following IDEs:

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

**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'				Yes

**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`

# setbuilddopt

---

**Purpose** Set active configuration build options

**Syntax** `IDE_Obj.setbuilddopt(tool,ostr)`  
`IDE_Obj.setbuilddopt(file,ostr)`

**IDEs** This function works with the following IDEs:

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

**Description** Use `IDE_Obj.setbuilddopt(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 `getbuilddopt` command to read a list of defined build tools.

If the VisualDSP++ and Code Composer Studio IDEs do not recognize the `ostr` argument, `setbuilddopt` sets 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.setbuilddopt(file,ostr)` to configure the build options for a file you specify with the `file` argument. The source file must exist in the active project.

**See Also** `activate`  
`getbuilddopt`

**Purpose** Program symbol table from IDE

**Syntax** `s = IDE_Obj.symbol`

**IDEs** This function works with the following IDEs:

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

**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.

<b>s Structure Field</b>	<b>Contents of the Specified Field</b>
<code>s(1).name</code>	String reflecting the symbol entry name.
<code>s(1).address(1)</code>	Address or value of symbol entry.
<code>s(1).address(2)</code>	Memory page for the symbol entry. For TI C6xxx processors, the page is 0.

For MULTI IDE, this table shows a few possible elements of `s`, and their interpretation.

<b>s Structure Field</b>	<b>Contents of the Specified Field</b>
<code>s(1).name</code>	String reflecting the symbol entry name.
<code>s(1).address</code>	Address or value of symbol entry.
<code>s(1).address</code>	Address or value of symbol entry in hex.

# symbol

---

You can use field address in `s` as the address input argument to read and write.

If you use `symbol` and the symbol table does not exist, `s` returns empty and you get a warning message.

Symbol tables are a portion of a COFF object file that contains information about the symbols that are defined and used by the file. When you load a program to the processor, the symbol table resides in the IDE. While the IDE may contain more than one symbol table at a time, `symbol` accesses the symbol table belonging to the program you last loaded on the processor.

## Examples

Build and load 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 CCS IDE

**Syntax**

```
IDE_Obj = ticcs  
IDE_Obj = ticcs('propertyname','propertyvalue',...)
```

**IDEs**

This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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 Embedded IDE Link creates the new object.

---

**Note** When `ticcs` creates the object `IDE_Obj`, it sets the working directory for the IDE to be the same as your MATLAB software working directory. When you create files or projects in the IDE, or save files and projects, this working directory 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
rt dx	'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
type	type	Defined types in the object	Void, Float, Double, Long, Int, Short, Char	List of the C data types in the project IDE_Obj accesses. Use add to include your C type definitions to the list

`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.

---

**Note** The output argument name you provide for `ticcs` cannot begin with an underscore, such as `_IDE_Obj`.

---

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 above, 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, as shown here:

```
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

IDE_Obj.type     % Returns information about the type object

Defined types : Void, Float, Double, Long, Int, Short, Char
```

## See Also

`ccsboardinfo`, `set`

---

<b>Purpose</b>	Set whether IDE window is visible while IDE runs
<b>Syntax</b>	<code>IDE_Obj.visible(state)</code>
<b>IDEs</b>	This function works with the following IDEs: <ul style="list-style-type: none"><li>• Analog Devices VisualDSP++</li><li>• Texas Instruments Code Composer Studio</li></ul>
<b>Description</b>	<p>Use <code>IDE_Obj.visible(state)</code> to make the IDE visible on the desktop or make it run in the background.</p> <p>To run the IDE in the background so it is not visible on the desktop, enter '0' for the <code>state</code> argument.</p> <p>To make the IDE visible on your system desktop, enter '1' for the <code>state</code> argument.</p> <p>You can use methods to interact with a IDE handle object, such as <code>IDE_Obj</code>, while the IDE is in both states, visible and not visible. You can interact with the IDE GUI while the IDE is visible.</p> <p>On the Microsoft Windows platform, if you make the IDE visible and look at the Windows Task Manager:</p> <ul style="list-style-type: none"><li>• While the IDE is visible (<code>state</code> is 1), the IDE appears on the <b>Applications</b> page of Task Manager, and the <code>IDE_Obj_app.exe</code> process shows up on the <b>Processes</b> page as a running process.</li><li>• While the IDE is not visible (<code>state</code> is 0), the IDE disappears from the <b>Applications</b> page, but remains on the <b>Processes</b> page, with a process ID (PID), using CPU and memory resources.</li></ul>
<b>Examples</b>	<p>In MATLAB, use the appropriate constructor function to create a IDE handle object for your IDE. The constructor function creates a handle, such as <code>IDE_Obj</code>, and starts the IDE.</p> <p>To get the visibility status of <code>IDE_Obj</code>, enter:</p>

# visible

---

```
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, do one of the following operations.

- 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 works with the following IDEs:

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

**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 (see below).

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).

To demonstrate how `write` uses *address*, here are some examples of the *address* argument:

<b>address Parameter Value</b>	<b>Description</b>
131082	Decimal address specification. The memory start address is 131082 and memory type is 0. This is the same as specifying [131082 0].
[131082 1]	Decimal address specification. The memory start address is 131082 and memory type is 1.
'2000A'	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify *address* as cell array, in which case you can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myaddress1{2} =  
'Program(PM) Memory';
```



```
myaddress2 myaddress2{1} = '20000'; myaddress2{2} =
'Program(PM) Memory';
```

```
myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem=write(...,datatype)` where the *datatype* argument defines the interpretation of the raw values written to DSP memory. The *datatype* argument specifies the data format of the raw memory image. The data is written starting from *address* without regard to data type alignment boundaries in the DSP. The byte ordering of the data type is automatically applied. The following MATLAB data types are supported:

<b>MATLAB Data Type</b>	<b>Description</b>
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 is 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` in the *MATLAB Function Reference*  
`read`

# writemsg

---

**Purpose** Write messages to specified RTDX channel

---

**Note** Support for writemsg on C5000 and C6000 processors will be removed in a future version.

---

**Syntax**

```
data = writemsg(rx,channelname,data)
data = writemsg(rx,channelname,data,timeout)
```

**IDEs** This function works with the following IDEs:

- Texas Instruments Code Composer Studio

**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`. Note again that this example works 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`

# xmakefilesetup

---

**Purpose** Configure Embedded IDE Link to generate makefiles

## Syntax

**IDEs** This function works with the following IDEs:

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

## Description

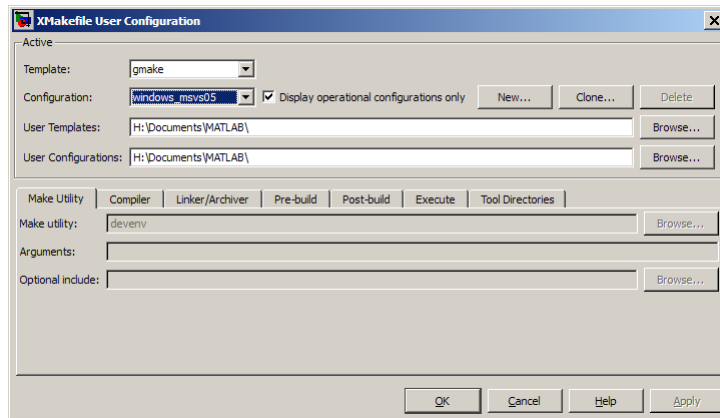
You can configure Embedded IDE Link to generate and build your software using makefiles. This process can use the software build toolchains, such as compilers and linkers, associated with the preceding list of IDEs. However, the makefile build process does not use the graphical user interface of the IDE directly.

Enter `xmakefilesetup` at the MATLAB command line to configure how Embedded IDE Link generates 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 Chapter 4, “Generating Makefiles”.

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 10-5, “Build action” on page 10-7

# xmakefilesetup

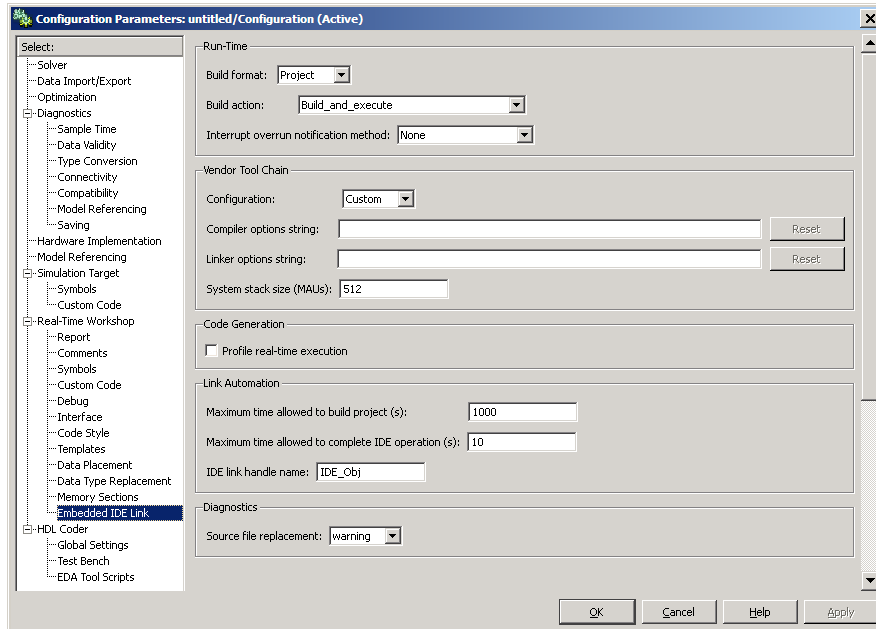
---



# Configuration Parameters

---

## Embedded IDE Link Pane



### In this section...

- “Overview” on page 10-4
- “Build format” on page 10-5
- “Build action” on page 10-7
- “Overrun notification” on page 10-10
- “Function name” on page 10-12
- “PIL block action” on page 10-13
- “Configuration” on page 10-15
- “Compiler options string” on page 10-17
- “Linker options string” on page 10-19
- “System stack size (MAUs)” on page 10-21

**In this section...**

“System heap size (MAUs)” on page 10-23

“Profile real-time execution” on page 10-24

“Profile by” on page 10-26

“Number of profiling samples to collect” on page 10-28

“Maximum time allowed to build project (s)” on page 10-30

“Maximum time allowed to complete IDE operations (s)” on page 10-32

“Export IDE link handle to base workspace” on page 10-33

“IDE link handle name” on page 10-35

“Source file replacement” on page 10-36

## Overview

Use this pane to configure the following aspects of the Embedded IDE Link software:

- 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.

## To get help on an option

- 1 Right-click the option's text label.
- 2 Select **What's This** from the popup menu.



## Build format

Defines how Real-Time Workshop 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.

---

**Note** PIL is a feature of the Real-Time Workshop Embedded Coder product. To use PIL in Embedded IDE Link, you must have a Real-Time Workshop Embedded Coder license.

---

## 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:** Build | Build\_and\_execute | Create\_project Archive\_library  
| Create\_processor\_in\_the\_loop\_project

**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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## Build action

Defines how Real-Time Workshop 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 Real-Time Workshop software to create a new project in the IDE.

#### 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 Real-Time Workshop 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”.

#### 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\_project 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic in the Embedded IDE Link User’s Guide.

### 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## PIL block action

Specifies whether Real-Time Workshop software builds the PIL block and downloads the block to the processor.

### Settings

**Default:** Create\_PIL\_block\_and\_download

Create\_PIL\_block\_build\_and\_download

Builds and downloads the PIL application to the processor after creating the PIL block. Adds PIL interface code that exchanges data with Simulink.

Create\_PIL\_block

Creates a PIL block, places the block in a new model, and then stops without building or downloading the block. The resulting project will not compile in the IDE.

None

Configures model to generate a IDE project that contains the PIL algorithm code. Does not build the PIL object code or block. The new project will not compile in the IDE.

### Tips

- When you click **Build** on the PIL dialog box, the build process adds the PIL interface code to the project and compiles the project in the IDE.
- If you select **Create PIL block**, you can build manually from the block right-click context menu.
- After you select **Create PIL Block**, *copy* the PIL block into your model to replace the original subsystem. Save the original subsystem in a different model so you can restore it in the future. Click **Build** to build your model with the PIL block in place.
- *Add* the PIL block to your model to use cosimulation to compare PIL results with the original subsystem results. Refer to the demo “Comparing Simulation and processor Implementation with Processor-in-the-Loop (PIL)” in the product demos Embedded IDE Link.

- When you select `None` or `Create_PIL_block`, the generated project will not compile in the IDE. To use the PIL block in this project, click **Build** followed by **Download** in the PIL block dialog box.

### Dependency

Enable this parameter by setting **Build action** to `Create_processor_in_the_loop_project`.

### Command-Line Information

**Parameter:** `configPILBlockAction`

**Type:** `string`

**Value:** `None` | `Create_PIL_block` |  
`Create_PIL_block_build_and_download`

**Default:** `Create_PIL_block`

### Recommended Settings

Application	Setting
Debugging	<code>Create_PIL_block_build_and_download</code>
Traceability	<code>Create_PIL_block_build_and_download</code>
Efficiency	<code>None</code>
Safety precaution	No impact

### See Also

For more information, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic in the Embedded IDE Link User’s Guide.

## 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 Code Composer Studio software to the generated project and code. The Compiler options string becomes `-g -d _DEBUG`

#### Release

Applies the Release project configuration defined by Code Composer Studio software to the generated project and code. Sets the **Compiler options** string to `-o2`.

### Dependencies

- Selecting Custom disables the reset options for **Compiler options string** and **Linker options string**.
- Selecting Release sets the **Compiler options string** to `-o2`.
- Selecting Debug sets the **Compiler options string** to `-g -d _DEBUG`

### 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.



## Compiler options string

Lets you enter a string of compiler options to define your project configuration.

### Settings

**Default:** No default

### Tips

- To import compiler string options from the current project in the IDE, click **Get from IDE**.
- To reset the compiler options to the default values, click **Reset**.
- 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 Embedded IDE Link software. **Custom** does not use any optimizations.
- Setting **Configuration** to **Debug** applies the `_Debug`, `-g`, and `-d` compiler flags defined by Code Composer Studio software.
- Setting **Configuration** to **Release** applies the IDE Release compiler options and adds the `-o2` optimization flag defined by Code Composer Studio software.

### 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## Linker options string

Enables you to specify linker command options that determine how to link your project when you build your project.

### Settings

**Default:** No default

### Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.
- To import linker string options from the current project in the IDE, click **Get from IDE**.
- To reset the linker command options to the default values, click **Reset**.

### 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## System stack size (MAUs)

Enter the amount of memory that is available for allocating stack data.

### Settings

**Default:** 8192

**Minimum:** 0

**Maximum:** Available memory

- Enter the stack size in minimum addressable units (MAUs)..
- 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 **Real-Time Workshop** pane to `idelink_ert.tlc` or `idelink_grt.tlc`, the software sets the **Maximum stack size** parameter on the **Optimization** pane to `Inherit` from `target` and makes it non-editable. In that case, the **Maximum stack size** parameter compares the value of  $(\text{System stack size}/2)$  with 200,000 bytes and uses the smaller of the two values.

### Command-Line Information

**Parameter:** `systemStackSize`

**Type:** `int`

**Default:** 8192

### Recommended Settings

Application	Setting
Debugging	<code>int</code>
Traceability	<code>int</code>

<b>Application</b>	<b>Setting</b>
Efficiency	int
Safety precaution	No impact

### **See Also**

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## System heap size (MAUs)

Allocates memory for the system heap on the processor.

### Settings

**Default:** 8192

**Minimum:** 0

**Maximum:** Available memory

- Enter the heap size in minimum addressable units (MAUs)..
- 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	<code>int</code>
Traceability	<code>int</code>
Efficiency	<code>int</code>
Safety precaution	No impact

### See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## 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 Embedded IDE Link 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics in the Embedded IDE Link User’s Guide..

## 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

For more information about PIL and its uses, refer to the “Verifying Generated Code via Processor-in-the-Loop” topic in the Embedded IDE Link User’s Guide.

For more information about using profiling, refer to the “profile” and “Profiling Code Execution in Real-Time” topics in the Embedded IDE Link User’s Guide..

## Number of profiling samples to collect

Specifies the number of profiling samples to collect. Collection stops when the buffer for profiling data is full.

### Settings

**Default:** 100

**Minimum:** 1

**Maximum:** Buffer capacity in samples

### Tips

- Data collection stops after collecting the specified number of samples. The application and processor continue to run.
- 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

<b>Application</b>	<b>Setting</b>
Efficiency	No impact
Safety precaution	No impact

**See Also**

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## 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 operations** timeout value.

### Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

### Command-Line Information

**Parameter:**TBD

**Type:** int

**Value:** Integer greater than 0

**Default:** 100

### Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

<b>Application</b>	<b>Setting</b>
Efficiency	No impact
Safety precaution	No impact

**See Also**

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## **Maximum time allowed to complete IDE operations (s)**

specifies how long 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:**TBD

**Type:** int

**Value:**

**Default:** 10

### **Recommended Settings**

<b>Application</b>	<b>Setting</b>
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact



## See Also

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## 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 Embedded IDE Link 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**

<b>Application</b>	<b>Setting</b>
Debugging	On
Traceability	On
Efficiency	No impact
Safety precaution	No impact

**See Also**

For more information, refer to the “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## 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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.

## Source file replacement

Selects the diagnostic action to take if Embedded IDE Link 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 Real-Time Workshop **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 “Embedded IDE Link Pane Parameters” topic in the Embedded IDE Link User’s Guide.



## A

- activate 9-2
- add 9-4
- address 9-7
- animate 9-15
- Archive\_library 3-44
- asynchronous scheduling 3-4

## B

- block limitations using model reference 3-45
- block recommendations 2-17
- blocks
  - use in target models 2-17
- blocks to avoid in models 2-17
- boards, selecting 3-3

## C

- C6000 model reference 3-43
- ccsboardinfo 9-17
- configuration parameters
  - pane 10-4
    - buildAction 10-7
    - buildFormat 10-5
    - Compiler options string: 10-17
    - configPILBlockAction 10-13
    - DiagnosticActions 10-36
    - Export IDE link handle to base
      - workspace: 10-33
    - Function name: 10-12
    - gui item name 10-28
    - IDE link handle name: 10-35
    - ideObjBuildTimeout 10-30
    - ideObjTimeout 10-32
    - Linker options string: 10-19
    - overrunNotificationMethod 10-10
    - Profile real-time execution 10-24
    - profileBy 10-26

- projectOptions 10-15
  - System heap size (MAUs): 10-23
  - System stack size (MAUs): 10-21
- configure 9-30
- configure the software timer 7-33
- connect to simulator 9-66
- CPU clock speed 7-33
- create custom target function library 3-42
- current CPU clock speed 7-33
- Custom Demo block 7-99
- custom source code 3-32

## D

- datatypemanager 9-33
- debug operation
  - new 9-119
- disable 9-50
- discrete solver 3-29

## E

- Embedded IDE Link™
  - build format 2-9
  - code generation options 2-9
- Embedded IDE Link™ software
  - introduction 1-2
- enable 9-56
- execution in timer-based models 3-9
- execution profiling
  - subsystem 5-13
  - task 5-11

## F

- file and project operation
  - new 9-119
- fixed-step solver 3-29
- flush 9-58

**G**

generate optimized code 2-9  
get symbol table 9-163

**H**

heap size, set heap size 2-12

**I**

IDE status 9-90  
Idle Task block 7-2  
info 9-70  
intrinsics. *See* target function library  
isenabled 9-80  
isreadable 9-82  
isrtdxcapable 9-87  
issues, using PIL 5-9  
isvisible 9-90  
iswritable 9-92

**L**

list 9-97  
list object 9-97  
list variable 9-97

**M**

manage data types 9-33  
MATLAB® API 1-5  
matrix, read from RTDX 9-134  
Memory Allocate block 7-5  
Memory Copy block 7-11  
model execution 3-4  
model reference 3-43  
    about 3-43  
    Archive\_library 3-44  
    block limitations 3-45  
    modelreferencecompliant flag 3-46  
    setting build action 3-44

Target Preferences blocks 3-45  
    using 3-44

model schedulers 3-4  
modelreferencecompliant flag 3-46  
msgcount 9-118

**O**

optimization, processor specific 2-9

**P**

PIL cosimulation  
    overview 5-3  
PIL issues 5-9  
processor configuration options  
    build action 2-9  
    overrun action 2-11  
processor function library. *See* target function library  
processor information, get 9-70  
processor specific optimization 2-9  
profiling execution  
    by subsystem 5-13  
    by task 5-11  
program file, reload 9-150  
project generation  
    selecting the board 3-3

**R**

read register 9-141  
readmat 9-134  
readmsg 9-137  
Real-Time Workshop solver options 3-29  
regread 9-141  
regwrite 9-146  
reload 9-150  
replacing generated code 3-32  
replacing linker directives 3-32  
RTDX



- isenabled 9-80
- isrtdxcapable 9-87
- message count 9-118
- read message 9-137
- readmat 9-134
- writemsg 9-180

RTDX channel, flush 9-58

RTDX message count 9-118

RTDX, disable 9-50

RTDX, enable 9-56

## S

- select blocks for models 2-17
- selecting boards 3-3
- set heap size 2-12
- set stack size 2-12
- set visibility 9-173
- simulator
  - connect to 9-66
- solver option settings 3-29
- source code replacement 3-32
- stack size, set stack size 2-12
- symbol 9-163
- symbol table, getting symbols 9-163
- synchronous scheduling 3-9

## T

- table of blocks to avoid in models 2-17
- target function library
  - assessing execution time after selecting a library 3-39

- create a custom library 3-42
- optimization 3-36
- seeing the library changes in your generated code 3-40
- selecting the library to use 3-38
- use in the build process 3-37
- using with link software 3-36
- viewing library tables 3-42
- when to use 3-38

Target Preferences blocks in referenced models 3-45

Target Preferences/Custom Board block 7-30

Target Support Package™

- create Simulink® model for targeting 2-16

TFL. *See* target function library

ticcs 9-165

timer, configure 7-33

timer-based models, execution 3-9

timer-based scheduler 3-9

timing 3-4

## V

view IDE 9-90

viewing target function libraries 3-42

visibility, setting 9-173

visible 9-173

## W

write register 9-146

writemsg 9-180