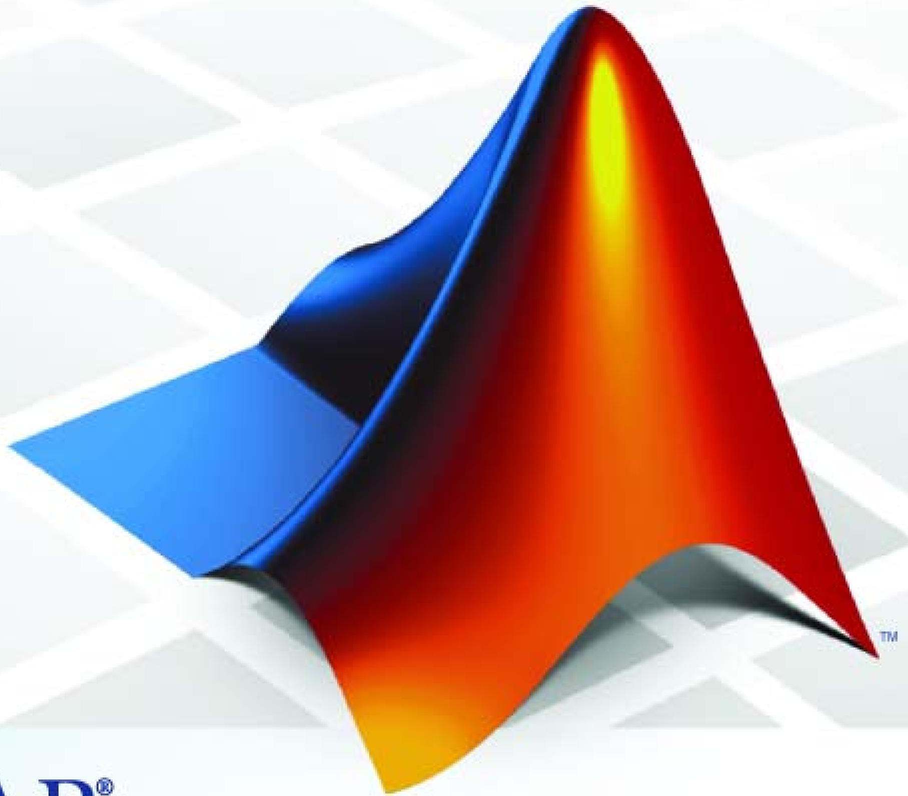
Embedded IDE Link™ 4
User's Guide

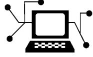
For Use with Altium® TASKING®



MATLAB® & SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

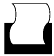
How to Contact The MathWorks



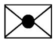
`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Embedded IDE Link™ User's Guide

© COPYRIGHT 2006–2010 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` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

May 2006	Online only	New for Version 1.0 (Release 2006a+)
September 2006	Online only	Version 1.0.1 (Release 2006b)
March 2007	Online only	Version 1.1 (Release 2007a)
September 2007	Online only	Revised for Version 1.2 (Release 2007b)
March 2008	Online only	Revised for Version 1.3 (Release 2008a)
October 2008	Online only	Revised for Version 1.3.1 (Release 2008b)
March 2009	Online only	Revised for Version 1.4 (Release 2009a)
September 2009	Online only	Revised for Version 4.0 (Release 2009b)
March 2010	Online only	Revised for Version 4.1 (Release 2010a)

Contents

1 Getting Started

Product Overview **1-2**
Introduction **1-2**
Project Generator **1-3**
Automation Interface **1-3**
Verification **1-4**
Optimization **1-5**

Software Requirements **1-6**

Supported Altium TASKING Toolsets **1-7**
Supported Versions **1-7**
Support for Other Versions **1-7**

Using This Guide **1-9**

Setting Target Preferences **1-10**
Procedure **1-10**
Target Preference Fields **1-13**

Working with Configuration Sets **1-16**
Adding the Embedded IDE Link Configuration Set Component **1-16**
Configuration Set Options **1-16**
Using Configuration Sets to Specify Your Target **1-19**
Setting Build Action **1-23**

Accessing Utilities for TASKING **1-27**
Embedded IDE Link Utilities for Use with TASKING dialog **1-27**
Tools Menu Items **1-29**

Option Sets **1-30**

What Are Option Sets? 1-30
Supported DAS Software 1-32

2 Components

Project Generator **2-2**
Overview of the Project Generator Component 2-2
Project-Based Build Process 2-4
Template Projects 2-4
Shared Libraries 2-6
Build Process — Folder Structure 2-9

Automation Interface **2-13**
Overview of Automation Interface Component 2-13
Classes 2-14
Using Objects 2-15
List of Methods 2-22
Details of Particular Methods 2-25

3 Verification

Processor-in-the-Loop (PIL) Cosimulation **3-2**
Processor-in-the-Loop Overview 3-2
PIL Workflow 3-3
Creating a PIL Block 3-5
Building, Running, and Debugging PIL Block Applications 3-8
PIL Metrics 3-11

C Code Coverage Reports **3-13**

Execution Profiling **3-15**
CrossView Pro Execution Profiling 3-15

Task Execution Profiling Kit for Real-Time Workshop Targets 3-18

Stack Profiling 3-19
What Is Stack Profiling? 3-19
PIL Applications 3-19
Non-PIL Applications 3-20
Infineon® TriCore Stack Depth Analyzer 3-21

Bidirectional Traceability Between Code and Model . . 3-22
Using Traceability 3-22
Enabling Traceability 3-23

MISRA C Rule Checking 3-25

4 Optimization

Compiler / Linker Optimization Settings 4-2

Target Memory Placement / Mapping 4-3

Execution and Stack Profiling 4-4
Execution Profiling 4-4
Stack Profiling 4-4

Target Specific Optimizations 4-5
C Language Extensions / Intrinsics 4-5
Target Optimized Libraries for Infineon XC166 and Infineon® TriCore 4-7
Target Optimized FIR / FFT Blocks for the Infineon® TriCore 4-8

Model Advisor 4-9

5 Tutorials

Tutorial: Using Option Sets **5-2**

Tutorial: Creating New Template Projects **5-4**
Creating New Template Projects **5-4**
Creating a New Configuration **5-7**

Tutorial: Configuring an Existing Model for Embedded IDE Link Software **5-9**

6 Configuration Parameters

Embedded IDE Link Pane **6-2**
Overview **6-3**
Build Action **6-4**
Target Preference Configuration **6-6**
Add build directory suffix **6-7**
Build directory suffix **6-8**
Export EDE handle to MATLAB base workspace **6-9**
EDE handle name **6-9**
Export CrossView Pro handle to MATLAB base workspace **6-11**
CrossView Pro handle name **6-11**
Configure model to build PIL algorithm object code **6-13**
PIL block action **6-14**

A Limitations and Tips

General Issues **A-2**
Problems with Installations in Read-Only Locations **A-2**
Simulink Configuration Set Reference Not Supported **A-2**

Serialization of Embedded IDE Link Objects Not Supported A-3

Debugger Issues **A-4**
ARM CrossView Pro Debugger Fails with File | Open Source Content A-4
On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware A-4

Build Process Issues **A-6**
Linker Errors Due to Limited Memory A-6
EDE Is Slow, Unresponsive, or Crashes A-7
Signal Processing Blockset Library Build Failures A-7
Memory Block Freed Twice Error A-8
8051 EDE Cannot Compile Files with Long Names A-8
DSP563xx Toolset Support Limitations A-8
"Create, Build and Execute Application Project" Build Action Fails A-9
C166 Toolset Warnings A-10
Build Error From Root Drive Location A-10
Limited Support for Nonfinite Values A-10
Memory Warning/Error Messages in the CrossView Pro Command Window When Using the Instruction Set Simulator A-13
C++ Code Generation Not Supported A-13
Video and Image Processing Blockset Library Not Supported A-14
Noninlined S-functions Calling rt_matrx.c Not Supported A-14
"Compiler optimization level" Configuration Parameter Has No Effect A-14
Configuration Changes Cause Build Errors With Referenced Models A-15

Processor-in-the-Loop Issues **A-16**
Generic PIL Issues A-16
On-Chip PIL Not Supported on ARM Hardware A-16
10-Second Pause on Termination of the CrossView Pro Debugger A-16
DSP563xx Link-Order Issue Can Cause PIL Application Failure A-17
No Support for TASKING Feature "Treat double as float" A-17

TASKING Optimization Settings May Cause Incorrect Cosimulation Results **A-18**

Issues Using Real-Time Workshop Software Without Real-Time Workshop® Embedded Coder Software . . **A-19**
Real-Time Workshop grt.tlc-Based Targets Not Supported for PIL **A-19**
"Save data to workspace" Causes Error **A-19**
DSP563xx Toolset Support Limitations **A-19**
Use ERT Target for Memory-Constrained Targets **A-20**
8051 GRT Limitations **A-20**

Index

1

Getting Started

- "Product Overview" on page 1-2
- "Software Requirements" on page 1-6
- "Supported Altium TASKING Toolsets" on page 1-7
- "Using This Guide" on page 1-9
- "Setting Target Preferences" on page 1-10
- "Working with Configuration Sets" on page 1-16
- "Accessing Utilities for TASKING" on page 1-27
- "Option Sets" on page 1-30

Product Overview

In this section...

"Introduction" on page 1-2
"Project Generator" on page 1-3
"Automation Interface" on page 1-3
"Verification" on page 1-4
"Optimization" on page 1-5

Introduction

Embedded IDE Link™ software lets you build, test, and verify automatically generated code using the MATLAB®, Simulink®, and Real-Time Workshop® products, and the Altium® TASKING® integrated development environment. Embedded IDE Link software makes it easy to verify code executing within the TASKING environment using a test harness model in Simulink. This processor-in-the-loop testing environment uses code automatically generated from Simulink models by the Real-Time Workshop® Embedded Coder™ product. A wide range of DSPs and 8-, 16- and 32-bit microprocessors and microcontrollers are supported including devices from the Infineon®, Renesas®, and Freescale™ product families. Embedded IDE Link software provides customizable templates for configuring hardware variants, automating MISRA C® code checking, and controlling the build process.

With Embedded IDE Link software, you can use MATLAB and Simulink to interactively analyze, profile and debug target-specific code execution behavior within TASKING software. In this way, Embedded IDE Link software automates deployment of the complete embedded software application and makes it easy for you to assess possible differences between the model simulation and target code execution results.

Embedded IDE Link software consists of a Project Generator component, an Automation Interface component, and features for code verification and optimization. The following sections summarize these components and features.

Project Generator

- 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 TASKING Toolsets provides a route to a large number of different target hardware platforms. Further customization is possible by using custom project templates, giving access to all options supported by the TASKING Toolset.

- Automated download and debugging

  Rapidly and effortlessly debug generated code in the CrossView Pro debugger, using either the instruction set simulator or real hardware.

Automation Interface

- MATLAB API for TASKING EDE (IDE)

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

  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.
  - Execute CodeWright API Library commands.

- MATLAB API for TASKING CrossView Pro (Debugger)

  Automate complex tasks in the TASKING CrossView Pro debugger by writing MATLAB scripts to communicate with the CrossView Pro application, or debug and analyze interactively in a live MATLAB session.

For example, you could

- Automate debugging by executing commands from the powerful CrossView Pro command language.
- Exchange data between MATLAB and the target running in the CrossView Pro application.
- Set breakpoints, step through code, set parameters and retrieve profiling reports

Verification

- Processor-in-the-loop (PIL) cosimulation

  Use cosimulation techniques to verify generated code running in an instruction set simulator or real target environment.

- C Code Coverage

  Use C code instruction coverage metrics from the CrossView Pro instruction set simulator during PIL cosimulation to refine test cases.

- Execution Profiling

  Use execution profiling metrics from the CrossView Pro instruction set simulator 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.

- Bi-Directional Traceability Between Model and Code

  Navigate to the generated code for a given Simulink block or, vice versa, to the Simulink block corresponding to a section of generated code.

- MISRA® Checker

  Use the TASKING compiler generated MISRA report to check for an appropriate level of MISRA compliance for your application.

Optimization

- Compiler / Linker Optimization Settings

  Use Template Projects to fully control compiler and linker optimization settings.

- Target Memory Placement / Mapping

  Use Template Projects to fully configure the target memory map.

- Execution Profiling

  Use execution profiling metrics from the CrossView Pro instruction set simulator 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.

- Target Optimized FIR / FFT Blocks for the Infineon® TriCore®

  Use example FIR / FFT blocks that call target optimized Infineon TriLib routines. These blocks can be over a hundred times faster than the regular blocks in the Signal Processing Blockset™ product. Additionally, create your own optimized blocks to provide more functionality.

Software Requirements

You must have the correct software installed. For example, the Simulink menu item **Tools > Utilities for Use with TASKING(R) IDE** is enabled only if Real-Time Workshop is installed in addition to MATLAB, Simulink, and Embedded IDE Link.

For detailed information about the software and hardware required to use Embedded IDE Link with Altium TASKING, refer to www.mathworks.com/products/ide-link/requirements.html and www.mathworks.com/products/ide-link/altium-adaptor.html.

Supported Altium TASKING Toolsets

Supported Versions

Embedded IDE Link software includes at least one reference template project for each supported toolset. The reference projects were created for specific versions of each Altium TASKING toolset and were used by The MathWorks for qualification testing. The supported toolset versions are:

- Infineon TriCore: TASKING VX-toolset for TriCore v2.5 r2

  See also “Regenerate Template Projects to Use Selected Toolset Versions” on page 1-8.
- Infineon® C166®: TASKING Tools for C166/ST10 v8.7 r1
- Renesas M16C: TASKING Tools for M16C v3.1 r1 patch 2
- ARM®: TASKING C Compiler for ARM v2.0 r2

  Simulator only, see “On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware” on page A-4.
- Freescale DSP563xx: TASKING Tools for DSP563xx v3.5 r3 patch 2
- 8051: TASKING Tools for 8051 v7.2 r1

The Renesas R8C family is supported by the Renesas M16C TASKING Toolset.

The Freescale DSP566xx family is supported by the Freescale DSP563xx Toolset.

Support for Other Versions

Check the Embedded IDE Link Product Support page for patches and additional toolchain version information.

For minor release increments it may be sufficient to create new default template projects. To do this,

1. Specify the location of your TASKING toolset in the Target Preferences (see “Setting Target Preferences” on page 1-10).

2 Close all projects/project spaces in the EDE, and close the EDE.

3 Move to a clean work folder or clean out the existing one.

4 Run the `tasking_generate_templates` command. You must specify your configuration description string, e.g.:

```
tasking_generate_templates('C166', true)
```

or

```
tasking_generate_templates('TriCore', true)
```

Note Make sure you check the Embedded IDE Link Product Support page for the latest information about toolchains qualified with the product. You may be able to obtain patches in order to use other toolsets.

Regenerate Template Projects to Use Selected Toolset Versions

The following toolsets should work after regenerating the template projects:

- TASKING VX-toolset for TriCore and PCP v2.5 r2
- As TASKING VX-toolset for TriCore v2.5 r2 but without On-Chip Debug Support (OCDS)
- "TASKING C/C++ Compiler for ARM v2.0 r2"

Some TASKING packages do not include On-Chip Debug Support (OCDS). For example, "TASKING C/C++, CrossView Pro SIM" does not include OCDS support, but "TASKING VX-Toolset" does. To use a package without OCDS support you must regenerate the template projects as previously described.

Using This Guide

To get started with Embedded IDE Link software:

1. Follow the instructions in "Setting Target Preferences" on page 1-10.

2. After you set target preferences, follow the instructions in "Working with Configuration Sets" on page 1-16 to see how to set up configurations using an example model.

3. Try the demos to gain experience using Embedded IDE Link software. Access the demos by selecting **Start > Links and Targets > Embedded IDE Link > Demos**.

4. See "Accessing Utilities for TASKING" on page 1-27 for a quick guide to the functionality available in the menus, with links to more information.

See the following chapters to learn about Embedded IDE Link software features:

- Chapter 2, "Components" explains the Embedded IDE Link software components: the Project Generator build process, and the Automation Interface objects.
- Chapter 3, "Verification" describes how to use PIL cosimulation and other product features for verification.
- Chapter 4, "Optimization" describes how to use product features for optimization.
- Chapter 5, "Tutorials" contains instructions to show you how to create new configurations and template projects, how to use Embedded IDE Link software with existing models, and how to use different build actions.

Setting Target Preferences

Procedure

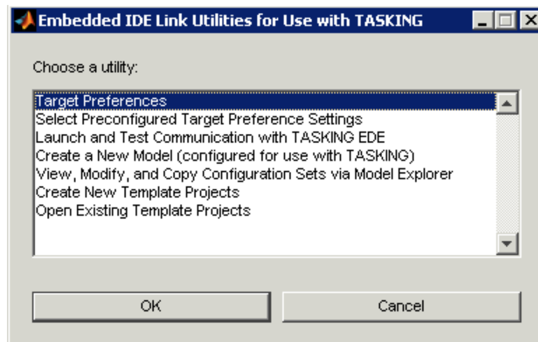
You must configure your target preferences to use Embedded IDE Link software.

Note Target preferences are persistent across MATLAB sessions. If you have used a previous version of Embedded IDE Link software, click **Reset to Default** before setting up your new preferences, to ensure you use the latest values for all fields.

---

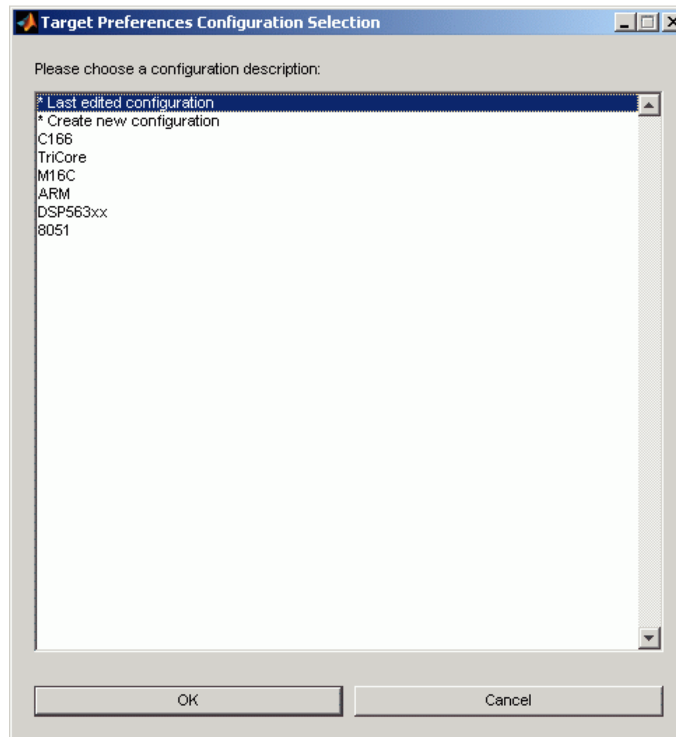
1. Enter `taskingutils` in the Command Window.

   The Embedded IDE Link Utilities for Use with TASKING dialog box appears.

2. Select `Target Preferences` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

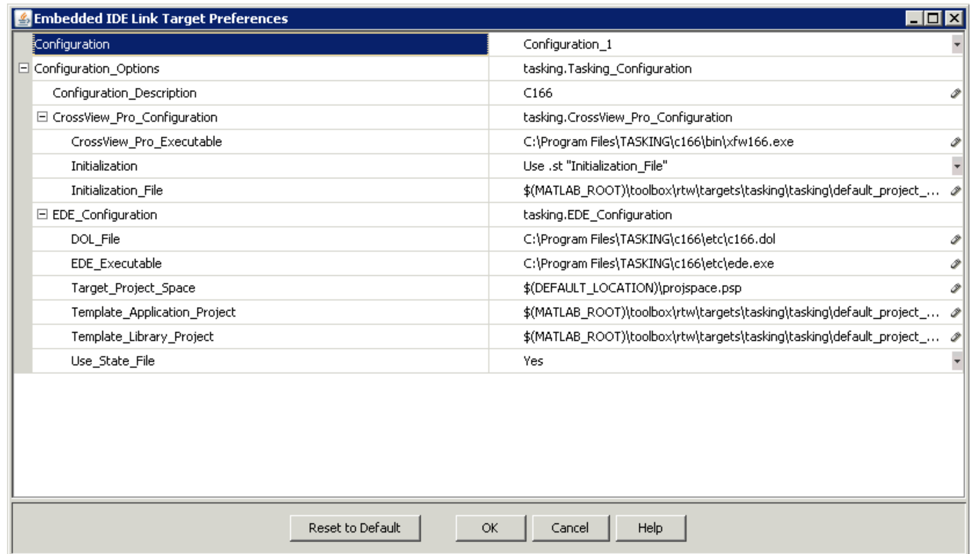


3 Select or create a configuration:

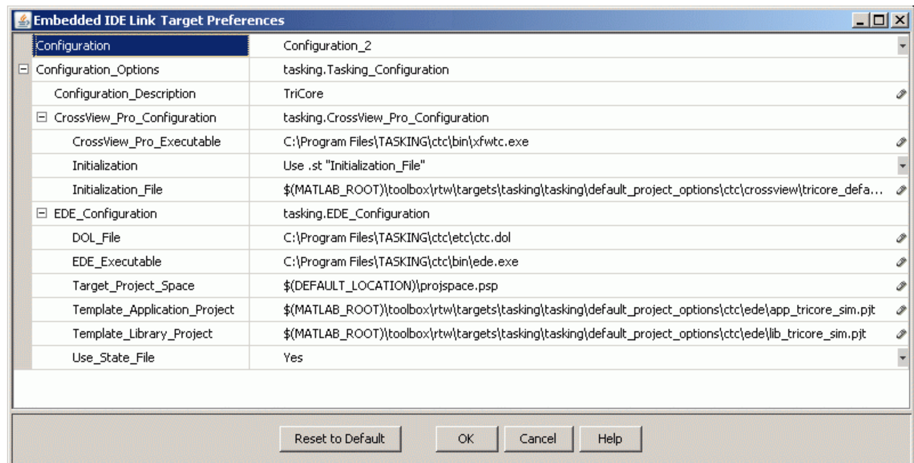
- Choose a predefined configuration from the list that matches your target.
- Alternatively, select `Create new configuration` to create a new configuration, and click **OK**. For new configurations, see the tutorial section "Creating a New Configuration" on page 5-7.

The Embedded IDE Link Target Preferences dialog box appears. You can use this dialog box to configure the location of your Altium TASKING toolchain executable and other files.

4 Click the plus to expand `Configuration Options`. Similarly, expand `CrossView_Pro_Configuration` and `EDE_Configuration`, as shown in the example. This example is set up for the `Infineon C166 Simulator` configuration.



5 Replace the string <ENTER_TASKING_PATH> to complete the path to the `CrossView_Pro_Executable`, the `DOL_File`, and the `EDE_Executable`. See the next section, “Target Preference Fields” on page 1-13, for details on each field. The following example is set up for the `Infineon TriCore Simulator` configuration.



If you have multiple configurations, you have to set them up in your target preferences only once, and then it is simple to switch between them. See the tutorial example "Working with Configuration Sets" on page 1-16.

6 Click **OK** to dismiss the Embedded IDE Link Target Preferences dialog box.

The next section explains each target preference field.

Target Preference Fields

Enter `tasking_edit_prefs` in the Command Window.

- `Configuration`

  Select a configuration from the drop-down list. There are preconfigured configurations for

  - C166
  - TriCore
  - M16C
  - ARM
  - DSP563xx
  - 8051

  If you have multiple configurations, you have to set them up in your target preferences only once, and then it is simple to switch between them. You can switch between them using this target preference field.

  Select a free configuration number to set up a new configuration from scratch. See "Creating a New Configuration" on page 5-7.

- `Configuration_Description`

  The title of the configuration. After it is created, this title is the name that appears in the **Target Preferences Configuration** drop-down list in the Configuration Parameters dialog box. Edit this field to change the name of the configuration. These names are predefined for the preconfigured configurations. For a new configuration enter a descriptive name (do not include spaces).

- `CrossView_Pro_Executable`

Enter the full path to your TASKING CrossView Pro installation to replace the string <ENTER_TASKING_PATH>. For example, for `Configuration_1` for `Infineon C166 Simulator`:

`D:\Applications\TASKING\c166\bin\xfw166.exe`

- `Initialization`

  This setting determines what the CrossView Pro Debugger executes when it first starts. There are three options.

  - `Use .st Initialization_File` This option is the default setting. "`.st`" files are in an internal file format used by The MathWorks to provide initialization options to CrossView Pro software during debugger start up. For example, a `.st` file may specify a CrossView Pro configuration file (`.cfg`) and target type for CrossView Pro to use. Each of the option sets shipped with Embedded IDE Link software specifies a corresponding `.st` file. For example, the `c166_sim` option set specifies the `c166_default.st` file, which includes basic initialization commands for the C166 CrossView Pro Simulator. See "Option Sets" on page 1-30 for related information. To customize your CrossView Pro configuration, you should use one of the `.ini` initialization options.
  - `Use .ini Initialization_File` Use this option if you have a custom `.ini` initialization file. The file should be a valid CrossView Pro initialization file for your custom configuration. Refer to your CrossView Pro application documentation for details.
  - `Use CrossView Pro Default .ini File` Use this option if you want to run CrossView Pro Default `.ini` file when launching the CrossView Pro Debugger. When launching CrossView Pro software you may be prompted to make configuration selections. Refer to your CrossView Pro application documentation to find the location of this `.ini` file, and for details of CrossView Pro initialization files.

- `Initialization_File`

  Full path of the initialization file corresponding to the `Initialization` field.

- `DOL_File`

  The full path to the TASKING EDE DOL file. For example, the `Infineon_C166_Simulator` Configuration has the <ENTER_TASKING

PATH>_\etc\c166.dol as the dol file. You need to replace <ENTER_TASKING_PATH> with your real TASKING installation path.

- EDE_Executable

  Enter the full path to your TASKING EDE installation to replace the string <ENTER_TASKING_PATH>. For example, for Configuration_1 for Infineon C166 Simulator, enter

  D:\Applications\TASKING\c166\bin\ede.exe

- Target_Project_Space

  When you build models, new projects in the TASKING EDE will be created. These projects belong to the project space defined in this entry. The default setting is $(DEFAULT_LOCATION)\projspace.psp. The code generation process expands the $(DEFAULT_LOCATION) token based on the build folder of the model, the model name, and model configuration settings, including the name of the template application project. You should avoid changing this default setting.

- Template_Application_Project

  When you build a Simulink model with Embedded IDE Link software, the generated projects for your application in the TASKING EDE have the same project settings as the template application project. This template project provides a centric place to manage the project options (e.g., compiler settings, linker settings, etc.) your Simulink models use during code generation. You can modify the project settings of the default template projects or create new ones. See "Accessing Utilities for TASKING" on page 1-27 for information on creating or opening template projects, and see "Template Projects" on page 2-4.

- Template_Library_Project

  The same as the Template_Application_Project field, but this is applicable for Library projects.

- Use_State_File

  Opens the TASKING EDE in its last saved state. For more information, refer to your TASKING EDE documentation.

Working with Configuration Sets

In this section...
“Adding the Embedded IDE Link Configuration Set Component” on page 1-16
“Configuration Set Options” on page 1-16
“Using Configuration Sets to Specify Your Target” on page 1-19
“Setting Build Action” on page 1-23

Adding the Embedded IDE Link Configuration Set Component

To add Embedded IDE Link configuration options to a model, select the menu item **Tools > Embedded IDE Link > Add Embedded IDE Link Configuration to Model**.

Similarly, you can use the menu item **Remove Embedded IDE Link Configuration from Model** to remove the configuration set component.

The following sections explain how to use the Embedded IDE Link configuration set component.

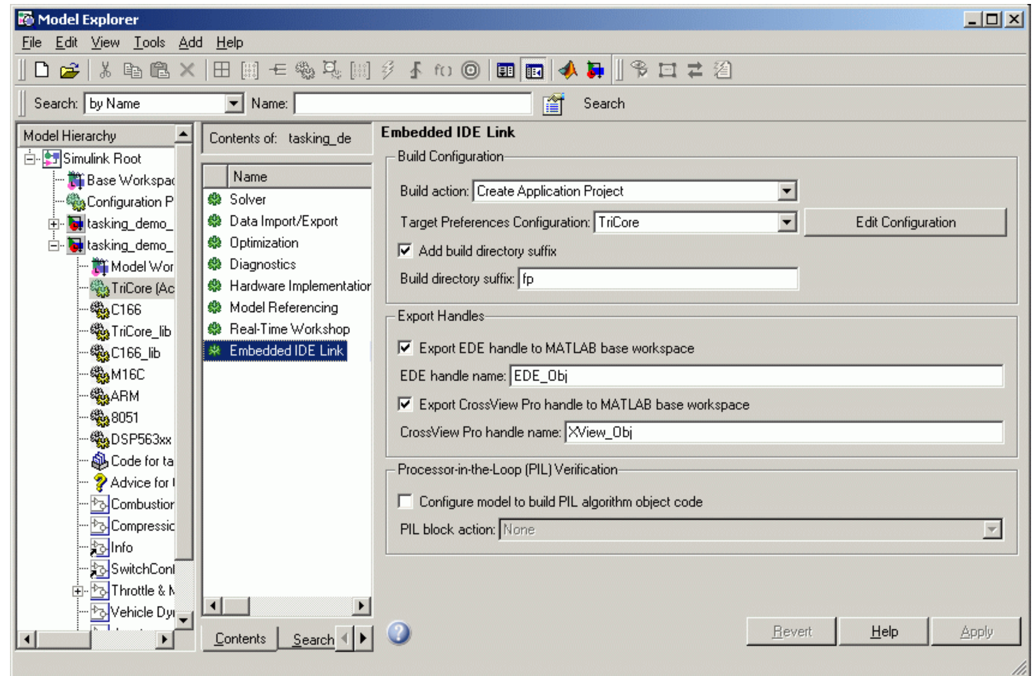
See also “Setting Up Configuration Sets” in the Simulink documentation for more information.

Configuration Set Options

To see Embedded IDE Link configuration options, navigate to the configuration parameters by any of the following paths:

- **Simulation > Configuration Parameters** in a model
- **Tools > Embedded IDE Link > Options** in a model
- **View > Model Explorer** in a model

Click **Embedded IDE Link** to see the following options.



The following options are available under **Build Configuration**:

- **Build action**

  Set what action to take after the Real-Time Workshop build process. You can create application and library projects in the Altium TASKING EDE and then stop, or you can also choose to build, execute, or debug. See "Setting Build Action" on page 1-23 for more details.

- **Target Preferences Configuration**

  Select target preference configurations. The names correspond to the `Configuration Description` for each configuration in the Target Preferences dialog box. Click **Edit Configuration** to open the Target Preferences dialog box for the currently selected configuration. See "Using Configuration Sets to Specify Your Target" on page 1-19.

- **Add build subdirectory suffix**

  Select the check box to specify a model-specific suffix to be added the regular Real-Time Workshop build folder suffix. This setting is useful to avoid shared utility function code generation errors which occur because of conflicts over Real-Time Workshop utility functions shared between different models.

  Clear this check box to use the default Real-Time Workshop build folder suffix. Not using an additional suffix may result in rebuilding shared libraries unnecessarily. See "Shared Libraries" on page 2-6 and particularly "Supporting Multiple Shared Utility Function Locations: Build Folder Suffix" on page 2-7 for details.

- **Build subdirectory suffix**

  Enter in the edit box a model-specific suffix to be added the regular Real-Time Workshop build folder suffix.

The following options are available under **Export Handles**:

- **Export EDE handle to MATLAB base workspace**

  Select this check box to export the EDE object handle to the workspace.

- **EDE handle name**

  Enter a MATLAB variable name for the exported handle.

- **Export CrossView Pro handle to MATLAB base workspace**

  Select this check box to export the CrossView Pro object handle to the workspace.

- **CrossView Pro handle name**

  Enter a MATLAB variable name for the exported handle.

See "Automation Interface" on page 2-13 for information on using these object handles.

The following options are available under **Processor-in-the-Loop (PIL) Verification**:

- **Configure model to build PIL algorithm object code**

Select this box to build PIL algorithm code.

- **PIL block action**

  Select one of the following PIL block actions

  - `Create PIL block, then build and download PIL application`

    Select this option to automatically build and download the PIL application after creating the PIL block. This option is the default when you select the option to configure the model for PIL.

  - `Create PIL block`

    Choose this to create the PIL block and then stop without building. You can build manually from the PIL block.

  - `None`

    Choose this to avoid creating a PIL block, for instance if you have already built a PIL block and do not want to repeat the action.

    See "Processor-in-the-Loop (PIL) Cosimulation" on page 3-2 for more information on using PIL settings.

Using Configuration Sets to Specify Your Target

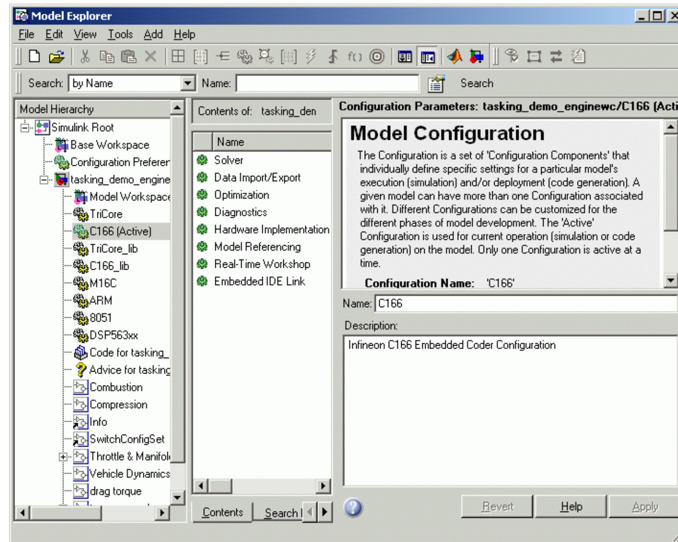
Follow the steps in this example to see where to find and change Embedded IDE Link software settings. These steps are described to help you find the settings you need to get started using the demo models. To use the demos, you need to specify your target by working with configuration sets.

This example describes how to use Embedded IDE Link software to build a project from a demo model using two different toolchains. The instructions refer to C166® and TriCore® TASKING toolchains; adapt the instructions to your toolchain as appropriate.

Finding the Embedded IDE Link Software Settings

1. Open the model `tasking_demo_enginewc`.

2. Double-click the `Active Configuration Set` block to open the Model Explorer (or select **View > Model Explorer**).



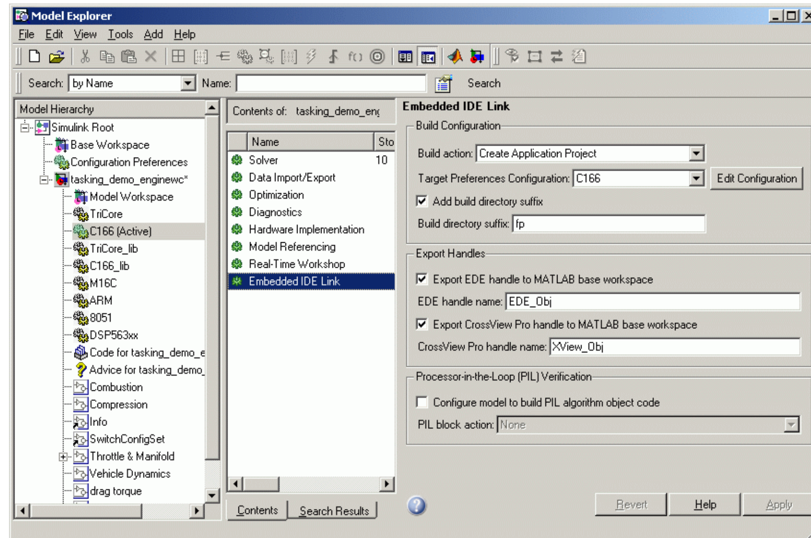
Under `TASKING_demo_enginewc` is a list of configuration sets you can review. The currently selected set is labeled `(Active)`.

Reviewing and Changing the Configuration Settings

Inspect the active configuration set.

1. The default active configuration set for this model is `C166`. If you want to use a different target, right-click the configuration set that matches your target, and select **Activate**.

2. Click **Embedded IDE Link** to see the configuration settings, as shown in the following figure.



3 The **Target Preferences Configuration** drop-down list shows all available target preference configurations. After you have set up target preferences for particular configurations, you can switch between them here (or in the Target Preferences dialog box).

a Click **Edit Configuration** to inspect your current target preferences.

b Before building, you must replace the string `<ENTER TASKING PATH>` to set up the correct paths to the target preferences `CrossView_Pro_Executable`, the `DOL_File`, and the `EDE_Executable`. See "Setting Target Preferences" on page 1-10.

c Click **OK** to dismiss the Target Preferences dialog box.

In the Embedded IDE Link demos, when you activate a configuration (e.g., `C166`), the appropriate **Target Preferences Configuration** is automatically selected. You may want to select a different target preference configuration description, e.g., if you have set up a custom configuration (such as `C167_user_hardware`). For an example, see "Creating a New Configuration" on page 5-7.

See "Adding the Embedded IDE Link Configuration Set Component" on page 1-16 for information on other Embedded IDE Link software settings in the Configuration Parameters.

4 Click **Real-Time Workshop** to see the selected system target file.

Note You can use a configuration set specifying any system target file with Embedded IDE Link software.

---

5 Click **Hardware Implementation** to see the C166 settings. If you are using a different target, make sure the settings match your device. Select from the **Device type** list. There are custom configurations and preconfigured settings that include the following processors:

- Infineon C16x, XC16x
- Infineon TriCore
- ARM 7/8/9
- Renesas M16C
- 8051 Compatible
- Freescale DSP563xx (16-bit mode)

Close the Model Explorer.

6 In the model `tasking_demo_enginewc`, right-click the `t_eng_speed` subsystem, and select **Real-Time Workshop > Build Subsystem**. Click **Build** in the dialog box to continue.

Watch the output messages in the MATLAB Command Window as code is generated, your TASKING toolchain EDE is launched, and a new project created.

Switching Target Preference Configurations

If you have multiple toolchains, you only have to set up your target preferences once. After this initial setup, it is simple to switch between different configurations. For example, to switch configurations from C166 to TriCore targets:

1 In the model `tasking_demo_enginewc`, double-click the `Active Configuration Set` block to open the Model Explorer.

2. Right-click `TriCore` and select **Activate**. Close the Model Explorer.

3. To rebuild the subsystem with the new settings, right-click the `t_eng_speed` subsystem, and select **Real-Time Workshop > Build Subsystem**.

   Watch the output in the MATLAB Command Window as code is generated, the TASKING C166 EDE is closed, the TASKING TriCore EDE is launched, and the new project created.

You can follow similar steps to specify your target in the other demo models. To view the demos, select **Start > Links and Targets > Embedded IDE Link > Demos**.

To switch between simulator and hardware implementations for the same target configuration, you can use option sets. See "Option Sets" on page 1-30.

The next section describes using the build action setting in this example.

Setting Build Action

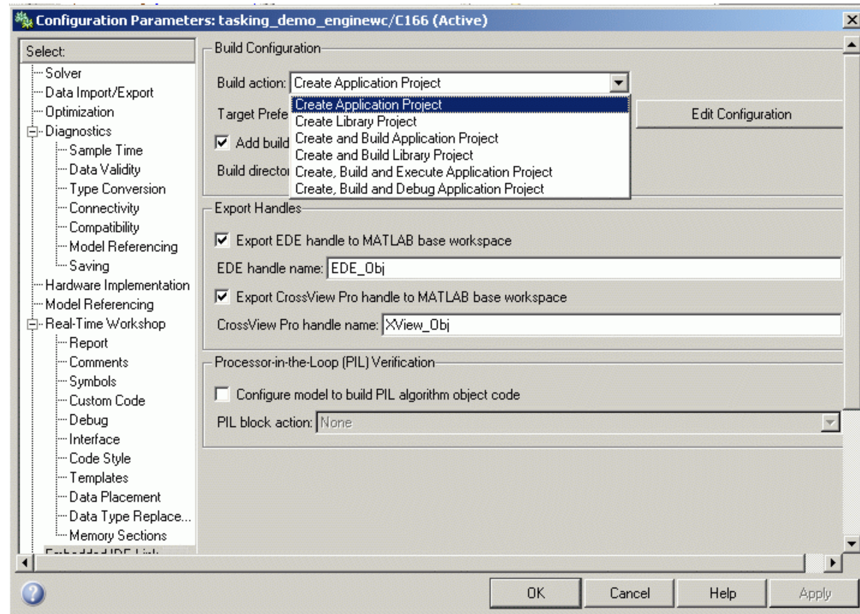
In this example, the model `tasking_demo_enginewc` is set up so the project is created but not built in the TASKING EDE.

To view this setting:

1. In the model `tasking_demo_enginewc`, select **Simulation > Configuration Parameters**.

2. Click **Embedded IDE Link** to see the **Build Configuration** parameters.

3. Look at the **Build Action** drop-down list.



Using this drop-down list, you can set what action to take after the Real-Time Workshop build process completes. You can create application and library projects in the TASKING EDE and then stop, or you can also choose to build, execute, or debug.

If you choose to build, execute, or debug, the CrossView Pro application will be launched.

Note The first time you build this model it will take a few minutes to compile the required Real-Time Workshop floating point library. This library is not rebuilt on subsequent builds unless necessary.

You can use the **Build Action** setting to do the following:

- `Create Application Project`

  Generates code for the model or subsystem, creates a TASKING application project for the selected TASKING configuration, connects to the TASKING EDE, and opens the application project (in addition to the required Real-Time Workshop and Signal Processing Blockset Library

projects, if required) in the TASKING EDE. This option does not build or execute the application.

An `EDE_Obj` object handle is exported to the MATLAB workspace (if the option **Export EDE handle to MATLAB base workspace** is selected). This object allows you to interact with the TASKING EDE from MATLAB. For more information, see the section on using object handles, "Automation Interface" on page 2-13.

Note To manually build the generated project in the TASKING EDE, right-click on the application project (starts with the same name as the model name), and select **Build**.

- `Create Library Project`

  Performs the same actions as `Create Application Project`, but this option archives the generated code into a library in the TASKING EDE. No `main.c` file is generated.

- `Create and Build Application Project`

  Performs the same actions as `Create Application Project`, but also instructs theTASKING EDE to build the application project.

Note To manually debug the executable from the application project, click the Debug Application icon in the TASKING EDE.

- `Create and Build Library Project`

  Performs the same actions as `Create Library Project`, but also instructs the TASKING EDE to build the Library project.

- `Create, Build and Execute Application Project`

  Performs the same actions as `Create and Build Application Project` and also downloads the executable file to your CrossView Target and runs the executable. No debugging information is downloaded into the target with this option.

A CrossView Pro object handle is exported to the MATLAB workspace (if the option **Export CrossView Pro handle to MATLAB base workspace** is selected). This object allows you to interact with the CrossView Pro debugger from MATLAB. For more information, see the section on using object handles, "Automation Interface" on page 2-13.

- `Create, Build and Debug Application Project`

  Performs the same actions as `Create, Build and Execute Application Project` but also downloads debugging information to the target. This option behaves the same way as the Debug Application icon in the TASKING EDE.

Accessing Utilities for TASKING

In this section...

"Embedded IDE Link Utilities for Use with TASKING dialog" on page 1-27

"Tools Menu Items" on page 1-29

Embedded IDE Link Utilities for Use with TASKING dialog

Open the Embedded IDE Link Utilities for Use with TASKING dialog box by entering `taskingutils` in the Command Window or double-clicking Launch TASKING Utilities in the Simulink block library.

You see the following options:.

- **Target Preferences**

  Opens the Target Preferences Configuration Selection dialog box, and after you choose a configuration to match your target (e.g., `TriCore`), you can edit the Target Preferences dialog box. In this dialog box, you can modify your TASKING preferences configurations. You can also open this dialog box from the MATLAB prompt by typing `tasking_edit_prefs`.

  You must set up your target preferences before you can use Embedded IDE Link software. See "Setting Target Preferences" on page 1-10.

- **Select Preconfigured Target Preference Settings**

  Opens the Target Preferences Configuration Selection dialog box. Choose a configuration to match your target and click **OK**. Then you can select a preconfigured option set. Your target preferences are automatically updated according to the option set you select, for example, specifying either hardware or simulator settings. See "Option Sets" on page 1-30.

- **Launch and Test Communication with TASKING EDE**

  Opens the Target Preferences Configuration Selection dialog box. Choose a configuration and click **OK**, and Embedded IDE Link software tests whether MATLAB can communicate successfully with the Altium TASKING EDE for the selected configuration. You see messages at the command line to confirm whether communication is successful.

- **Create a New Model (configured for use with TASKING)**

  Creates a new untitled Simulink model, with Embedded IDE Link configuration set options already added. You can also configure an existing model by selecting the Simulink model menu item **Tools > Utilities for Use with TASKING(R) IDE > Add Embedded IDE Link Configuration to Model**.

- **View, Modify, and Copy Configuration Sets via Model Explorer**

  Opens the Model Explorer where you can edit all configuration sets available for each currently open model.

- **Create New Template Projects**

  The Embedded IDE Link product ships with preconfigured application and library template projects for the default configurations in the Target Preferences dialog box. You might, however, create your own template projects (using preconfigured options as a starting point), and use them with any configuration. See “Tutorial: Creating New Template Projects” on page 5-4 for an example, and “Template Projects” on page 2-4 for more information.

  This option opens the Target Preferences Configuration Selection dialog box. Choose a configuration and click **OK**, and Embedded IDE Link software launches the appropriate TASKING EDE and creates new template projects for a specific **Tasking Configuration**. When you are prompted, choose a project folder, a template name, and an option set. See “Option Sets” on page 1-30 for more details. `app_template_name.pjt` and `lib_template_name.pjt` are created for the configuration you selected.

- **Open Existing Template Projects**

  Opens existing application and library template projects in the TASKING EDE for the selected **Tasking Configuration**. You can modify these options; however, it is preferable to do this by first creating new template projects, which avoids overwriting the default template projects. If you modify the default template projects, you can use the following function to recreate the defaults: `tasking_generate_templates`. You must specify your configuration description string, e.g.: `tasking_generate_templates('C166', true)`.

Note Opening or editing template projects causes the regeneration of application and library projects. When making any changes to template projects, it is important to make sure your changes are saved. To do this, remove the project from the project space; otherwise the changes may not be applied immediately. To remove a current project from the project space, right-click on it and choose **Remove from Project Space**.

- **Demos**

  Opens the Embedded IDE Link Demos page in the Help browser.

Tools Menu Items

In a Simulink model, you can access Embedded IDE Link menu items in the **Tools** menu. Select **Tools > Utilities for Use with TASKING(R) IDE** to see the following submenu items.

- **Target Preferences**

  As it does in the **Start** menu, this menu choice opens the Target Preferences Configuration Selection dialog box. After you choose a configuration, you can edit the Target Preferences Setup dialog box. You must set up your target preferences before you can use Embedded IDE Link software. See “Setting Target Preferences” on page 1-10.

- **Add Embedded IDE Link Configuration to Model**

  Adds Embedded IDE Link configuration options to the model configuration parameters.

  To see exactly which configuration parameter settings are changed, refer to `tasking_addto_configset.m`. Enter `edit tasking_addto_configset`.

- **Remove Embedded IDE Link Configuration from Model**

  Removes Embedded IDE Link configuration options from the model’s configuration parameters.

- **Options**

  Opens the Configuration Parameters dialog box to show Embedded IDE Link software options. See “Configuration Set Options” on page 1-16.

Option Sets

In this section...
"What Are Option Sets?" on page 1-30
"Supported DAS Software" on page 1-32

What Are Option Sets?

Option sets are preconfigured settings to specify the target configuration for the Altium TASKING tools. For example, after you set up your target preferences for a TriCore configuration, you can use option sets to switch between using an instruction set simulator configuration, two hardware board configurations, or a simulator with some MISRA C rule checking.

You can use option sets either:

- To switch between default target configurations, or
- When creating new template projects, to set up an initial configuration that you can choose to modify later

See "Tutorial: Using Option Sets" on page 5-2 for instructions.

The following preconfigured option sets are available.

A notation of "*" indicates the default in the Target Preferences. The processor type for the default configurations below is defined by your TASKING toolchain.

- Infineon TriCore:
  - * `tricore_sim`: Default instruction set simulator configuration.
  - `tricore_sim_misra`: As `tricore_sim`, but with some example MISRA C rule checking enabled. See also the TriCore MISRA C demo example, `tasking_demo_misra.m`, with instructions under Embedded IDE Link Demos.
  - `tricore_1796b`: Infineon TriCore 1796b hardware configuration.
  - `tricore_1766b`: Infineon TriCore 1766b hardware configuration.

- Infineon C166:
  - `c166_sim` : Default instruction set simulator configuration.
  - `c167cr` : Phytec kitCON-C167CR serial connection to hardware (`_hw`) and simulator (`_sim`) configurations.
  - `*c167cs` : Phytec phyCORE-C167CS serial connection to hardware (`_hw`) and simulator (`_sim*`) configurations.
  - `st10f252` : STMicrolectronics MB449 ST10F25x EVA Board serial connection to hardware (`_hw`) and simulator (`_sim`) configurations.
  - `st10f269` : Phytec phyCORE-ST10F269 serial connection to hardware (`_hw`) and simulator (`_sim`) configurations.
  - `xc164cm` : Infineon XC164CM U CAN start kit USB connection to hardware (`_hw_u_can`) and simulator (`_sim_u_can`) configurations. See "Supported DAS Software" on page 1-32.
  - `xc167ci`: On-board parallel port wiggler connection to the Infineon XC167CI Starter Kit hardware (`_hw`) and simulator (`_sim`) configurations.

    `xc167ci_hw_usb` : USB wiggler connection to the XC167CI

    Note For `xc167ci` targets, you must change jumper 501 when switching between USB wiggler and on-board parallel port wiggler. See your board manual for details.

- Renesas M16C
  - `* m16c_sim`: Default instruction set simulator configuration.
  - `r8ctiny_sim`: Renesas R8C Tiny instruction set simulator configuration.
- ARM:
  - `* arm_sim`: Default instruction set simulator configuration.
  - `arm_sim_big_endian`: As arm_sim, but in big-endian mode.
- Freescale DSP563xx:
  - `* dsp563xx_sim`: DSP563xx Family, 16-bit memory model, instruction set simulator configuration.

  - dsp566xx_sim: DSP566xx Family instruction set simulator configuration.

- 8051:
  - * i8051_sim: Default, large memory model, no language extensions, floating point, instruction set simulator configuration.

Supported DAS Software

For the XC164CM and certain TriCore hardware like TC1766 and TC1796, you need to download and install the supported DAS software. If your installation of the TASKING toolset did not come with DAS, then you can download the latest DAS software from this URL:

http://www.infineon.com/das.

At the time of writing, the latest tested DAS versions are:

- DAS Edition v2.6.2
- JTAG JDRV LPT Server v2.4.0

Make sure you restart your computer as instructed after DAS installation.

Components

- "Project Generator" on page 2-2
- "Automation Interface" on page 2-13

Project Generator

In this section...

"Overview of the Project Generator Component" on page 2-2
"Project-Based Build Process" on page 2-4
"Template Projects" on page 2-4
"Shared Libraries" on page 2-6
"Build Process — Folder Structure" on page 2-9

Overview of the Project Generator Component

The Embedded IDE Link Project Generator Component provides a customizable build process that is designed to work with the highly customizable code generation process provided by Real-Time Workshop. See "Project Generator" on page 1-3 for a summary.

To explain the separation of duties between Real-Time Workshop and Embedded IDE Link, the following sections discuss the terms *code generation process* and *build process*.

Code Generation Process

The code generation process is performed by the Real-Time Workshop family of products and is the process of translating a Simulink model into C code.

Customized code generation, perhaps to create target-specific device drivers or target-optimized code, is often a key requirement for users who want to generate code from Simulink models.

Real-Time Workshop and Real-Time Workshop Embedded Coder provide a variety of mechanisms for users to customize the code generation process. For example, the standard code generation process, using the regular system target files (like `grt.tlc` and `ert.tlc`) can be customized by making changes to the model's configuration parameters. Alternatively, for an even greater level of customization, including the ability to define custom Real-Time Workshop options, you can use a user created system target file.

The demos that come with Embedded IDE Link make use of the first type of customization with regular system target files. That is, the standard code generation process has been tailored for the appropriate target platform simply by changing the model's configuration parameters.

For greater flexibility, you should use a custom system target file. For further details on customizing the code generation process, see the Real-Time Workshop and Real-Time Workshop Embedded Coder documentation.

Build Process

The build process is performed by Embedded IDE Link and is the process of taking the C code produced by the code generation process and building (assembling, compiling, and linking) it for the target platform.

A customized build process, perhaps to use optimized compiler and linker settings, or perhaps to produce a MISRA compliance report, is often a key requirement for users wishing to build code produced from Simulink models.

Embedded IDE Link provides access to the full build process customization capabilities of the TASKING tools by allowing the user to set up the exact required configuration in TASKING. Embedded IDE Link then uses this configuration as a template for the build process.

Memory Placement Example

As an example, to consolidate the descriptions above of code generation and the build process, consider the common task of placing program data into a particular area of memory on a target platform.

Usually, this is achieved by using compiler-specific notations (like #pragmas) to define special memory sections and to assign data definitions to those sections. Additionally, a linker command file defines the different available memory regions on the target, and where in these regions the different memory sections should be located.

Splitting this task between the processes of code generation and building could be done as follows:

1 Customized code generation defines memory sections and assigns data.

2. Customized build process defines memory regions and assigns memory sections.

Project-Based Build Process

The Embedded IDE Link build process automatically creates TASKING EDE projects representing the application and libraries to be built.

A Real-Time Workshop application usually consists of some application code that makes references to modules that are part of libraries like the Real-Time Workshop library. Another common library is the Signal Processing Blockset library, used with the Signal Processing Blockset.

Embedded IDE Link creates separate projects for the main application code and each required library. The required libraries are included in the main application projects as subprojects.

Although the build process is project-based, underlying the projects are "makefiles" that can be used independently of the EDE. For an example of how to obtain the appropriate make command, see the demo instructions in `tasking_demo_objects.m` .

Target Project Space

Embedded IDE Link places projects in a project space known as the *target project space*. The location of the target project space is controlled by the `Target_Project_Space` setting in the Target Preferences, and usually depends on the `$(DEFAULT_LOCATION)` token, which is expanded based on the current folder at the time the build process is invoked, the model name, and model configuration settings, including the name of the template application project.

Template Projects

Template projects are regular TASKING EDE projects that are used by Embedded IDE Link to allow customization of the build process. Template projects are tied to particular TASKING Configurations as set up in the Target Preferences.

There are two types of template projects: application, and library template projects.

The application template project is used as the template for application projects and the library template project is used as the template for library projects.

Relocation of Template Projects

During the build process, the template project is copied to a target project location, and is then populated with the information relating to how to build the generated code.

Therefore, the project options of the template project become the project options of the target project, and hence the build process is customized according to the template project.

On subsequent build processes, Embedded IDE Link determines whether the template project has been updated since it was last copied to the target project location. If it has, then the target project is updated with a new copy of the template project. Otherwise, the target project is not updated from the template project.

Note Project options should be updated in the template project and not in the target project.

How the Build Process Modifies the Relocated Template Project

The Embedded IDE Link build process determines if any changes (preprocessor defines, include paths and source files) to the target project are required to build the code associated with a particular model, and updates the target project only if required. Thus, unnecessary project rebuilding is avoided.

Any include paths and preprocessor defines in the template project are always maintained in the target project. Maintaining this information is useful for keeping the include path to the compiler's standard header files, and setting global defines.

Additionally, the optional startup code file automatically generated by the EDE is also maintained.

Note Adding any other source files to your template project is not supported and will result in errors. Instead, you should add source files to the project by adding them to the Real-Time Workshop Build Info object by using either the Real-Time Workshop Custom Code settings in the configuration parameters, the `rtwmakecfg.m` mechanism, or by writing your own post code generation command (taking care not to overwrite any existing commands). See the Real-Time Workshop documentation for details.

Shared Libraries

Embedded IDE Link models that share the same target project space share required libraries such as the Real-Time Workshop library. Sharing of libraries means that a library is only built the first time a model that requires it is built.

The advantages of this shared library approach are

- No unnecessary per-model building of libraries; models with similar library requirements (e.g., integer code only) can share libraries.
- Libraries are built with the project options specified in the corresponding template project.

- Multiple sets of libraries, each with custom model, project options, or both can coexist.

Utility Function Generation: Shared Location

The shared library approach uses the Real-Time Workshop "Utility Function Generation" feature.

By setting utility function generation to use a shared location, rather than the model-specific default, you can ensure that the library projects created have no dependence on model-specific generated code. This feature is the key to allowing library projects to be shared between models.

As an example, consider the generated header file, `rtwtypes.h`, that contains the set of Real-Time Workshop data types available for compiling code modules, including any libraries.

With the utility function generation set to the default, individual `rtwtypes.h` files are generated into each code generation folder. Therefore, multiple definitions of `rtwtypes.h` would exist for a library shared between these models. The problem is, how can one of these `rtwtypes.h` files be chosen to build the library?

Setting the utility function generation to use a shared location provides a solution. In this case, a single `rtwtypes.h` file is generated into a folder shared between a set of models. This single file can be used to build the library without any dependence on the model-specific generated code.

Supporting Multiple Shared Utility Function Locations: Build Folder Suffix

The approach outlined in the previous section works well for a single set of models that have the same shared utility requirements.

However, what happens if you have two sets of models, each set with different shared utility requirements?

Normally, the Real-Time Workshop code generation process uses the current working folder as the location for generated files. In this location, it supports

only a single shared utilities folder for each system target file. Therefore, it is possible for conflicts over the contents of the shared utility folder to occur.

Example 1. For example, conflicts would occur if the Hardware Implementation settings were different for two models using the same system target file. If the standard `grt.tlc` or `ert.tlc` code generation process is customized by changing configuration set parameters, this situation is highly likely to occur.

To work around this problem, when using a `Target_Project_Space` (specified in the Target Preferences) containing the `$(DEFAULT_LOCATION)` token, Embedded IDE Link automatically appends the name of the current template application project to the regular Real-Time Workshop build folder suffix. This creates code generation and project folders that are specific to the current template application project, and so also specific to the current Hardware Implementation settings. Different Hardware Implementation settings always have different template projects.

Example 2. Another common example of this conflict, for two models sharing the same system target file, would be if one model was configured to support floating-point numbers and the other was configured to support integer code only.

To work around this conflict, use the Embedded IDE Link options **Add build subdirectory suffix** and **Build subdirectory suffix**.

If you select the **Add build subdirectory suffix** check box, then the **Build subdirectory suffix** you enter is appended to the regular Real-Time Workshop build folder suffix (before the name of the template application project discussed earlier, see “Template Projects” on page 2-4). This creates code generation and project folders that are specific to both the **Build subdirectory suffix** setting and the template projects.

For example, you can add `fp` for floating point models and `int` for non-floating-point models.

Note Using the same build subfolder suffix for a similar set of models allows them to generate code into their own working folder, avoiding conflict with other models, while still allowing a shared utilities folder.

This feature of Embedded IDE Link removes the need for the user to manually manage changing folders to avoid shared utility folder conflicts.

See the demo models for examples of using this setting: Embedded IDE Link for use with Altium TASKING Demos.

Build Process — Folder Structure

The following table shows the typical folders that are created, relative to the current working folder, during the Real-Time Workshop code generation process and the Embedded IDE Link build process.

Folder	Contents

`$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)\pjt_$(MODEL)`
e.g,
`ert_rtw_int_tricore_sim\pjt_fuelsys0`
	Main project: `$(MODEL).pjt` and associated files.

`$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)\pjt_rtwlib`	Real-Time Workshop library project: `rtwlib.pjt` and associated files.
`$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)\pjt_rtwshared` (if required)	Shared utilities library project: `rtwshared.pjt` and associated files.

`$(MODEL)_$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)`
e.g.,
`fuelsys0_ert_rtw_int_tricore_sim`
	Real-Time Workshop code generation folder.

Key	
$(MODEL)	Real-Time Workshop code generation model name (e.g., fuelsys0).
$(TEMPLATE_NAME)	Token expanded from the name of the template application project in the target preferences (e.g., tricore_sim). If the project name is prefixed with “app_” this token is removed.
$(REG_SUFFIX)	Regular Real-Time Workshop build folder suffix (e.g., ert_rtw).
$(MODEL_SUFFIX)	Model-specific build folder suffix (e.g., int).

See the next section, “Command Line Project Information” on page 2-10, for details about finding file names, paths, and other build information.

Command Line Project Information

When you build an application you can see information containing links at the MATLAB command line. You can use these links to get further details such as paths to projects, preprocessor defines, include paths, added files and their locations.

The following example shows a typical output:

```
### Building the PIL Application...
### Updating EDE projects according to BuildInfo object.
Please wait...
Creating project: t_shift_alg_ert_rtw_pil.pjt
Updating preprocessor defines in project:
t_shift_alg_ert_rtw_pil.pjt
Updating include paths in project:
t_shift_alg_ert_rtw_pil.pjt
Adding source files to project:
t_shift_alg_ert_rtw_pil.pjt
```

You can click the hyperlinks within these messages to get more information. The build messages are more readable with this information hidden, and the links provide access when you require more details.

Click the project file name (e.g., t_shift_alg_ert_rtw_pil.pjt) to see the full path to the project being built, like the following example:

```
Project path: D:\MATLAB\\work\tricore_fp\tricore_sim\
pjt_t_shift_alg_ert_rtw_pil\t_shift_alg_ert_rtw_pil.pjt
```

Click preprocessor defines to see a list of preprocessor defines similar to the one in the following example:

```
t_shift_alg_ert_rtw_pil.pjt preprocessor defines:

INTEGER_CODE=0
MAT_FILE=0
MODEL=t_shift_alg
MT=0
MULTI_INSTANCE_CODE=0
NCSTATES=0
NUMST=1
ONESTEPFCN=1
TERMFCN=1
TID01EQ=0
```

Click include paths to see a list of include paths similar to the one in the following example:

```
t_shift_alg_ert_rtw_pil.pjt include paths:

$(PRODDIR)\include
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw
D:\MATLAB\work\tricore_fp
D:\MATLAB\matlab\toolbox\rtw\targets\tasking\taskingdemos
D:\MATLAB\matlab\extern\include
D:\MATLAB\matlab\simulink\include
D:\MATLAB\matlab\rtw\c\src
D:\MATLAB\matlab\rtw\c\libsrc
D:\MATLAB\matlab\rtw\c\ert
D:\MATLAB\work\tricore_fp\slprj\ert\_sharedutils
D:\MATLAB\matlab\toolbox\rtw\targets\tasking\tasking\pil
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw_pil
```

Click source files to see a list of files added and their full paths.

```
t_shift_alg_ert_rtw_pil.pjt added files:

D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface.h
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_common.h
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_lib.c
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_lib.h
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\
tasking_pil_main.c
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw_pil\
pil_interface.c
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw_pil\
pil_interface_data.h
D:\MATLAB\work\tricore_fp\tricore_sim\
pjt_exp_t_shift_alg_ert_rtw\exp_t_shift_alg_ert_rtw.pjt
D:\MATLAB\work\tricore_fp\tricore_sim\pjt_rtwlib\rtwlib.pjt
```

Automation Interface

In this section...

"Overview of Automation Interface Component" on page 2-13

"Classes" on page 2-14

"Using Objects" on page 2-15

"List of Methods" on page 2-22

"Details of Particular Methods" on page 2-25

Overview of Automation Interface Component

The Embedded IDE Link Automation Interface Component provides powerful MATLAB API's for automating interaction with the TASKING EDE and CrossView Pro Debugger. See "Automation Interface" on page 1-3 for a summary.

Objects for Embedded IDE Link

Embedded IDE Link uses object-oriented programming techniques and requires a basic knowledge of some object-oriented terminology. The following are some fundamental terms you should understand:

- Object — Something you can operate on. An object is an instance of a class, created by calling the class constructor.
- Class — A class defines the properties and methods common to all objects of the class.
- Constructor — A function that creates an object, based on the class definition, and initializes it.
- Method — An operation on an object, defined as part of the class definition.
- Property — Part of an object, treated as a variable at times, that is defined as part of the class definition.
- Handle — A mechanism to access any object that Embedded IDE Link creates. Used in this guide to refer to the object. Often the handle is the name you assign when you create the object.

The following sections describe how to use and get help for Embedded IDE Link objects. See "Objects Demo Example" on page 2-22 for an example demonstrating some basic capabilities of Embedded IDE Link objects.

Classes

The following table shows the different classes that are provided with Embedded IDE Link for use with Altium TASKING.

Class	Description
`tasking.edeapi`	Represents the TASKING EDE.
`tasking.edeprojectspace`	Represents a project space in the TASKING EDE.
`tasking.edeproject`	Represents a project in the TASKING EDE.
`tasking.xviewapi`	Represents the TASKING CrossView Pro debugger.
`tasking.Tasking_Configuration`	Property of a `tasking.edeapi` class representing TASKING configuration details.
`tasking.EDE_Configuration`	Property of a `tasking.tasking_Configuration` representing EDE configuration details.
`tasking.CrossView_Pro_Configuration`	Property of a `tasking.tasking_Configuration` representing CrossView Pro configuration details.

Using Objects

The topics in this section are:

1 “Using the altiumtasking Function” on page 2-15

2 “Creating an Object Directly” on page 2-19

3 “Determining the Available Methods for a Class” on page 2-20

4 “Obtaining Help for a Class Method” on page 2-20

5 “Calling a Method” on page 2-21

6 “Determining the Available Properties for a Class” on page 2-21

7 “Accessing a Property” on page 2-21

8 “Objects Demo Example” on page 2-22

Using the altiumtasking Function

You can use `altiumtasking` to create Embedded IDE Link objects for Altium TASKING. The command

```
[ede,xview,projspc,proj] = altiumtasking
```

returns the following results:

- *ede* — A handle to the TASKING EDE application.
- *xview* — A handle to the TASKING CrossView Pro application.
- *projspc* — A handle to the project space currently open in the EDE.
- *proj* — A handle to the active project in the EDE.

Note You can only run `altiumtasking` after you configure your target preferences. See “Setting Target Preferences” on page 1-10 in the Embedded IDE Link for Use with Altium TASKING documentation.

When you first run `altiumtasking`, the Target Preferences Configuration Selection dialog box opens. Select your required configuration (for example, `TriCore`), and click **OK**.

- If you supplied a valid project space file when you configured the target preferences, `altiumtasking` opens that project space in the EDE window and returns a handle to it. However, if no valid project space file exists, then `altiumtasking` returns an empty project space handle.
- If one or more projects are defined in the project space, and opened, `altiumtasking` returns a handle to the active project. If no project is opened, `altiumtasking` returns an empty project handle.

You can also run `altiumtasking` in the following ways:

`[`*`ede`*`,`*`xview`*`,`*`projspc`*`] = altiumtasking` does not create and return the project handle.

`[`*`ede`*`,`*`xview`*`] = altiumtasking` does not create and return the project space and project handles.

`ede` `= altiumtasking` does not create and return the CrossView Pro application, project space and project handles.

`altiumtasking(`*`Property`*`, `*`Value`*`)` enables you to specify parameters that control the behavior of the function. For example:

```
altiumtasking('configDesc','TriCore')
```

specifies the use of the `TriCore` configuration in the function.

Create EDE and CrossView Pro Handles. This example generates EDE and CrossView Pro handles.

```
>>[ede, xview] = altiumtasking;

Registering COM object: "D:\share\apps\BuildTools\win32\TASKING\apps\tricore\v2.5r2\ctc\bin\x
Creating COM object: xfwtc.CommandLine
Initializing COM object: -G "C:\Temp" -ini "C:\Temp\tricore_default.ini" -tcfg "D:\share\apps

Testing COM communications with CrossView Pro by sending command: "echo MATLABLinkTest"
```

```
[Test timeout is 60 seconds, to allow hardware setup - use "Ctrl-C" to terminate]
Test successful.
```

Note You must save preferences to the CrossView Pro `.ini` file. The link software overwrites the temporary file, (`C:\Temp\tricore_default.ini`, when it creates a new `xviewapi` object.):

1. In CrossView Pro, select **File > Exit**. The Options dialog box opens.
2. On the **Save** tab, select the **Save desktop and target settings** check box.
3. Click **Exit**. CrossView Pro exits after a few moments, saving your preferences in the `.ini` file.

You can view the handles CrossView Pro created.

```
>> ede
tasking.edeapi (handle)
  configuration: [1x1 tasking.Tasking_Configuration]

 "configuration" property:
  Configuration_Description: 'TriCore'
         EDE_Configuration: [1x1 tasking.EDE_Configuration]
  CrossView_Pro_Configuration: [1x1 tasking.CrossView_Pro_Configuration]

  Connected to EDE: 0

>> xview
tasking.xviewapi (handle)
 configuration: [1x1 tasking.CrossView_Pro_Configuration]

  "configuration" property:
    CrossView_Pro_Executable: [1x76 char]
          Initialization: [1x29 char]
       Initialization_File: [1x107 char]
```

```
Status:
 Current executable: None
   Current project: None
 isRunning: 0
 eventReporting: 1
```

Create an EDE handle for TriCore . This example creates an EDE handle for `TriCore`.

```
>> obj = altiumtasking('configDesc', 'TriCore')

tasking.edeapi (handle)
 configuration: [1x1 tasking.Tasking_Configuration]

 "configuration" property:
     Configuration_Description: 'TriCore'
             EDE_Configuration: [1x1 tasking.EDE_Configuration]
   CrossView_Pro_Configuration: [1x1 tasking.CrossView_Pro_Configuration]

   Connected to EDE: 0
```

Creating an Object Directly

Embedded IDE Link allows you to create objects directly. To find out how to create an object of a particular class you can use the `help` function to find help for the constructor. At the MATLAB command prompt, enter

```
help <classname>.<constructorname>
```

For example, for the `tasking.edeapi` class, enter

```
help tasking.edeapi.edeapi
```

For the `tasking.edeprojectspace` class, enter

```
help tasking.edeprojectspace.edeprojectspace
```

Follow these steps to create example objects.

1. To create a `tasking.edeapi` object, you call the constructor as follows:

   ```
   Ede = tasking.edeapi
   ```

   The name on the left side of the "=" could be any valid MATLAB identifier and is the handle to the object.

   You must choose a configuration, then communication is tested with the TASKING EDE. At the command line you see the configuration target preferences.

2. To create a `tasking.edeprojectspace` object, you call the constructor as follows:

   ```
   tasking.edeprojectspace(projspace, edeapi)
   ```

   where `projspace` is the absolute path of the TASKING Project Space this object will relate to, and `edeapi` is a `tasking.edeapi` object, as shown in the following example:

   ```
    ps = tasking.edeprojectspace('D:\MATLAB\work\
   myprojspace.psp', Ede)
   ```

3. To create a `tasking.edeproject` object, you call the constructor as follows:

   ```
   tasking.edeproject(proj, edeprojspace)
   ```

where `proj` is the absolute path of the TASKING Project this object relates to, and `edeapiprojspace` is a `tasking.edeprojectspace` object, as shown in the following example:

```
proj = tasking.edeproject('D:\MATLAB\work\myproj.pjt', ps)
```

4 To create a `tasking.xviewapi` object, you call the constructor as follows

```
xv = tasking.xviewapi
```

You must choose a configuration, then communication is tested with CrossView Pro. At the command line, you see the configuration target preferences.

Determining the Available Methods for a Class

After you create an object, you can find the available methods by running the "methods" function.

1 For example, to find the methods available on the `tasking.edeapi` object created above (in "Creating an Object Directly" on page 2-19), enter `methods(Ede)`.

2 To find the methods available on the `tasking.edeprojectspace` object previously created, enter `methods(ps)`.

3 To find the methods available on the `tasking.edeproject` object previously created, enter `methods(proj)`.

4 To find the methods available on the `tasking.xviewapi` object previously created, enter `methods(xv)`.

To see the methods available, refer to the tables in "List of Methods" on page 2-22.

Obtaining Help for a Class Method

To get help for a class method, you can use the `help` function.

For example, to find out more about the `getProject` method of the `tasking.edeapi` class, you could enter the following command:

```
help tasking.edeapi.getProject
```

MATLAB returns the following output:

```
GETPROJECT - get the active Project in the EDE
project = getProject
project: edeproject object representing the active Project
 in the EDE
project will be empty if there is no open project
```

To see the methods available, refer to the tables in “List of Methods” on page 2-22.

Calling a Method

When you know the details of a class method, you can call it using dot (.) notation.

For example, to get a `tasking.edeproject` object representing the active project, run the following command:

```
project = Ede.getProject
```

Determining the Available Properties for a Class

After you create an object, you can find the available properties by running the `get` function.

For example, to find the properties available on the `tasking.edeapi` object created above, enter

```
get(Ede)
```

Accessing a Property

You can access a property of a class using dot (.) notation.

For example, to get the “configuration” property of the `tasking.edeapi` object created above, enter:

```
config = Ede.configuration
```

```
tasking.Tasking_Configuration (handle)
      Configuration_Description: 'C166'
              EDE_Configuration: [1x1 tasking.EDE_Configuration]
    CrossView_Pro_Configuration: [1x1 tasking.CrossView_Pro_
Configuration]
```

Objects Demo Example

For experience using objects, you can work through the demo example, `tasking_demo_objects.m`, found under Embedded IDE Link for use with Altium TASKING Demos.

This example provides step-by-step instructions for using Embedded IDE Link objects to communicate with the TASKING EDE and CrossView Pro debugger from the MATLAB command line. You can use any command available in the powerful CrossView Pro command language. The demo illustrates using objects during the process of building and debugging projects.

List of Methods

See the following tables for lists of available methods:

- “Methods for Class tasking.edeapi” on page 2-22
- “Methods for Class tasking.edeprojectspace” on page 2-24
- “Methods for Class tasking.edeproject” on page 2-24
- “Methods for Class tasking.xviewapi” on page 2-24

The public methods are shown in the tables (methods beginning with “`p`” or “`p_`” are private methods and should not be used).

Methods for Class tasking.edeapi

close	getOptionSetNames
disp	getProject
display	getProjectSpace
edeapi	getTargetProject

exec	getToolchainInfo
execApiMacro	newProject
execRetNumeric	newProjectSpace
execRetString	newProjectTemplates
getCreatedEDEProcess	newProjectTemplatesViaUI
getOptionSet	newTempProjectSpaceIfNoneOpen
openProjectTemplates	processTemplateProject
pwd	validateToolchainDirectory
hilite_system	connect
isconnected	

Methods for Class tasking.edeprojectspace

add	deleteParentDir
getEDE	isopen
checkValid	disp
getOriginalPath	new
checkValidProject	display
getPath	open
close	edeprojectspace
isequal	remove

Methods for Class tasking.edeproject

add	getEDE	isopen
build	getFiles	new
checkValid	getHyperlink	open
close	getIncludes	rebuild
debug	getMakeCmd	remove
disp	getOriginalPath	run
display	getPath	setCDefines
edeproject	getProjectSpace	setIncludes
getBuildOutput	getTarget	setPerformToolchainName-Check
getCDefines	hasFile	
getDir	isequal	

Methods for Class tasking.xviewapi

addBreakpointCallback	getEventReporting
getFunctionConfiguration	debug

disp	halt
removeBreakpointCallbacks	display
isRunning	setEventReporting
downloadAndRun	execute
xviewapi	executeAndWait
getCommandResponse	getExecutable
getProject	hilite_system
readMemoryUnits	writeMemoryUnits
reset	stackProfile
stackProfileReset	

Details of Particular Methods

The following methods of the `tasking.xviewapi` object simplify reading from and writing to target memory units (the smallest addressable unit in the memory of the target).

- `readMemoryUnits`

  To see help for this function, enter

  ```
  help tasking.xviewapi.readMemoryUnits
  ```

  at the MATLAB command line.

- `writeMemoryUnits`

  To see help for this function, enter

  ```
  help tasking.xviewapi.writeMemoryUnits
  ```

  at the MATLAB command line.

Use these functions with the MATLAB functions, `typecast` and `swapbytes`, for reading and writing data of different datatypes.

To see examples of syntax, see the demo example, `tasking_demo_objects.m`, found under Embedded IDE Link for use with Altium TASKING Demos.

3

Verification

- "Processor-in-the-Loop (PIL) Cosimulation" on page 3-2
- "C Code Coverage Reports" on page 3-13
- "Execution Profiling" on page 3-15
- "Stack Profiling" on page 3-19
- "Bidirectional Traceability Between Code and Model" on page 3-22
- "MISRA C Rule Checking" on page 3-25

Processor-in-the-Loop (PIL) Cosimulation

In this section...

"Processor-in-the-Loop Overview" on page 3-2
"PIL Workflow" on page 3-3
"Creating a PIL Block" on page 3-5
"Building, Running, and Debugging PIL Block Applications" on page 3-8
"PIL Metrics" on page 3-11

Processor-in-the-Loop Overview

Overview of PIL Cosimulation

Processor-in-the-loop (PIL) cosimulation is a verification technique designed to help you evaluate how well a candidate algorithm (e.g., a control system) operates on the actual target processor selected for the application.

When using Embedded IDE Link software, you have the following options for PIL cosimulation:

- Top-model PIL simulation mode — you can run a complete model as a PlL simulation on your target processor or instruction set simulator.
- Model block PIL simulation mode — use PIL cosimulation for a model reference component.
- PIL block — you can create a PIL block from one of several Simulink components including a model, a subsystem in a model, or subsystem in a library.

For information on processor-in-the-loop and how to use these different options, see"Verifying Compiled Object Code with Processor-in-the-Loop Simulation" in the Real-Time Workshop Embedded Coder documentation:

Note All options (i.e. Top-model PIL, Model block PIL, and PIL block) are available when you use Embedded IDE Link software with Altium TASKING tools.

To learn how to use the top-model and Model block PIL simulation modes, refer to the Real-Time Workshop Embedded Coder documentation linked above.

The following sections describe demos and detailed information on using the **PIL block** with Embedded IDE Link software and Altium TASKING tools.

PIL Workflow

You can work through the PIL block verification workflow demo for a hands-on example illustrating using SIL and PIL for system and unit testing: `tasking_demo_system_simulation.mdl`.

By running this demo you will learn how to:

- Use Software-in-the-Loop (SIL) to verify correct behavior of source code, generated by Real-Time Workshop Embedded Coder software and executing on the host processor
- Use Processor-in-the-Loop (PIL) to verify correct behavior of object code and generate metrics; the object code is cross-compiled from source code generated by Real-Time Workshop Embedded Coder software and executes on a target embedded processor
- Create system and unit test models
- Work with multiple heterogeneous target processors
- Include existing / legacy algorithms for SIL and PIL verification
- Export a generated algorithm for inclusion in an existing project
- Generate a fully deployable model-based application

Using target_block_verify

The function target_block_verify is used in the PIL Verification Workflow demo, tasking_demo_system_simulation.mdl.

You can use target_block_verify to verify a generated PIL or SIL block and compare the results with the simulation or algorithm block.

```
[LOG_SIGS1, LOG_SIGS2] = target_block_verify('BLOCK1', 'BLOCK2')
```

turns on signal logging for the outports of BLOCK1, the model containing BLOCK1 is simulated and the logged signals are returned in LOG_SIGS1.

Next, BLOCK1 and BLOCK2 are swapped, the same model is simulated again, and the logged signals for BLOCK2 are returned in LOG_SIGS2.

To verify a SIL or PIL block, set BLOCK1 to the simulation or algorithm block, and set BLOCK2 to the generated PIL or SIL block for BLOCK1.

Use full path names of Simulink blocks for BLOCK1 and BLOCK2.

BLOCK2 may be in the same model as BLOCK1, or in its own model. The model(s) containing BLOCK1 and BLOCK2 are loaded.

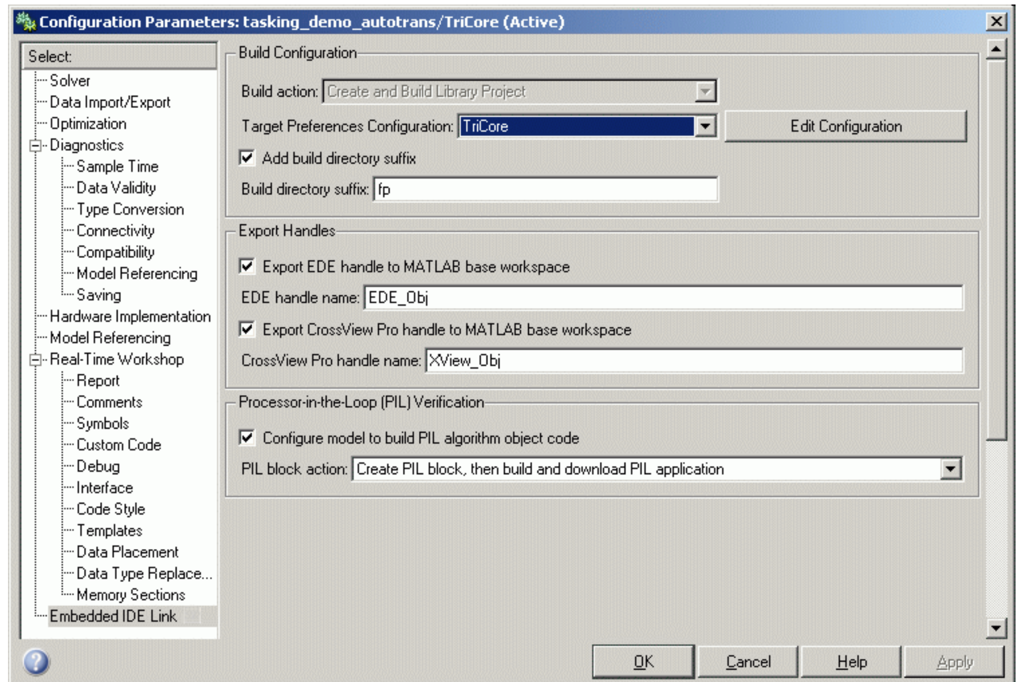
LOG_SIGS1 and LOG_SIGS2 are ModelDataLogs objects containing all the logged signals for the outports of BLOCK1 and BLOCK2 respectively. The data returned for each outport is a Timeseries object that allows comparison and plotting capabilities.

If BLOCK1 and BLOCK2 are in the same model, then one LOG_SIGS output is returned containing the data for both BLOCK1 and BLOCK2.

Caution target_block_verify makes temporary changes to the model by swapping BLOCK1 and BLOCK2 in addition to setting some logging options. Although target_block_verify restores the original settings of the model, it is recommended that you save a copy of your model first.

Creating a PIL Block

The PIL settings can be found in the Configuration Parameters dialog box under the **Embedded IDE Link** settings.



The following options are available under **Processor-in-the-Loop (PIL) Verification**

- **Configure model to build PIL algorithm object code**

  Select this box to create PIL algorithm object code as part of the Real-Time Workshop code generation process.

- **PIL block action**

  Select one of the following PIL block actions:

  - `Create PIL block, then build and download PIL application`

Select this option to automatically build and download the PIL application after creating the PIL block. This option is the default when you select the option to configure the model for PIL.

- `Create PIL block`

  Choose this option to create the PIL block and then stop without building. You can build manually from the PIL block.

- `None`

  Choose this option to avoid creating a PIL block, for instance if you have already built a PIL block and do not want to repeat the action.

After you create and build a PIL block, you can either:

- Copy it into your model to replace the original subsystem (save the original subsystem in a different model so it can be restored), or
- Add it to your model to compare with the original subsystem during cosimulation.

See "Building, Running, and Debugging PIL Block Applications" on page 3-8 for more details.

Building, Running, and Debugging PIL Block Applications

This section includes the following topics:

- "Building and Downloading PIL Applications" on page 3-8
- "PIL Debugging" on page 3-10

Building and Downloading PIL Applications

After you create a PIL block, you must build and download it before you can use it for cosimulation. You can use the **PIL Block Action** setting in the Configuration Parameters to automatically build and download the PIL application after the PIL block is created (select `Create PIL block, then build and download PIL application`). If you choose not to use this option, you can use the PIL block to build and download manually.

To build and download the PIL application manually:

1. Double-click the PIL block to open the mask.
2. Click **Build**. Wait until the **Application** name in the mask is updated and you see the "build complete" message.
3. Click **Download**.
4. Wait until the output in the MATLAB command window stops and you see the "download complete" message in the PIL block, and then click **OK** to close the block mask.

The PIL Application is now ready. To cosimulate with it, you must copy the PIL block into your model, either to replace the original subsystem or in addition to it for comparison. Click **Start Simulation** to run a PIL cosimulation.

After the test, Embedded IDE Link software returns execution profiling, code coverage, and stack profiling reports to MATLAB for your review. See "PIL Metrics" on page 3-11 for more information.

Note When copying PIL blocks to be used in the same model or in different models that simulate simultaneously, you must click the **Download** button in the PIL block mask in the new block after copying.

Clicking **Download** creates new connections (handles) to the TASKING EDE and CrossView Pro debugger. Otherwise, the same debugger handle may be used by multiple PIL blocks simultaneously and cosimulation errors or incorrect results may occur. This concern does not apply when copying PIL blocks created automatically as part of the build process because the untitled model and test harness are typically not simulated together.

See the Embedded IDE Link demos for examples with instructions to enable you to build and download PIL blocks and use them in cosimulation.

PIL Block Parameters. For generic PIL block information, see “Setting Up PIL Simulations With the Embedded IDE Link Product PIL Block” in the Real-Time Workshop Embedded Coder documentation.

Embedded IDE Link software creates PIL blocks with both the **Simulink system path** and **Configuration** properties automatically configured.

The available **Configurations** correspond to the **TASKING Configuration** descriptions in the Target Preferences.

Some guidelines for choosing a valid configuration:

1. The configuration must generate debugging information because Embedded IDE Link software requires this information to communicate with the PIL application.

2. The configuration must be compatible with the **Target Preferences Configuration** that was used to build the PIL algorithm. The fact that these two configurations need not match exactly allows the flexibility for the PIL algorithm to be compiled as if for a production environment, for example, without generating debugging information. However, you must be careful to ensure that the configurations are compatible in terms of linking, otherwise build errors occur when building the PIL application. In many cases, it is appropriate to use exactly the same configuration for

building both the PIL algorithm and PIL application and therefore no errors can ever occur because of incompatibilities between configurations.

PIL Debugging

Prior to PIL cosimulation you can use the CrossView Pro debugger to set breakpoints, so that you can step through the code and watch variables during cosimulation. To do this, you must set breakpoints in CrossView Pro prior to starting the cosimulation as follows:

1. When the build process completes, a minimized CrossView Pro window should appear on your Windows Start menu. Maximize the CrossView Pro window.

2. In CrossView Pro, select **File > Open Source**, and choose a source file to open. A typical choice would be to open the main generated file associated with the algorithm, e.g. `model.c`.

3. Choose a location in the file to set a breakpoint and click the "breakpoint" button to the left of the line. A typical location for setting a breakpoint in the `model.c` file would be one of the `step` functions.

Note You can set multiple breakpoints in multiple files if you wish.

---

4. To add a variable to the watch, double-click the variable, and then click **Add Watch** in the Expression Evaluation window. A typical variable to add to the watch would be either the external inputs or external outputs structures, which usually represent all of the inputs and outputs of the algorithm.

5. Start the PIL cosimulation in Simulink. When the breakpoint is hit, Simulink pauses. CrossView Pro is available for debugging, and watch variables are updated. You can step through the code, set more breakpoints, and analyze data.

6. When you are finished debugging, you can continue running by clicking the "play" button in CrossView Pro. This will allow the PIL cosimulation to continue. If you left the breakpoint in place then the cosimulation stops

at that point again. To return to uninterrupted cosimulation, remove the breakpoints.

Caution Never remove the PIL synchronization breakpoint (set on the `pilDataBreakpoint` function). This breakpoint is used to maintain synchronization between Simulink and CrossView Pro.

As an alternative to manual configuration in CrossView Pro, you can obtain a handle to the `tasking.xviewapi` object associated with a PIL block by using the `tasking_pil_crossview_handle` command as follows:

```
crossview = tasking_pil_crossview_handle('block')
```

where `block` is the full Simulink system path to the PIL block. You can use `gcb` to obtain the system path after clicking on the PIL block.

This handle can be used prior to PIL cosimulation to configure breakpoints, etc., by using the CrossView Pro command language.

Caution This handle should not be used during PIL cosimulation as this could lead to incorrect PIL results or termination of the PIL cosimulation.

10-Second Pause on Termination of the CrossView Pro Debugger. When terminating an instance of the CrossView Pro debugger application that was launched by Embedded IDE Link software, there is a pause of about 10 seconds before the CrossView Pro window closes. This 10-second pause is the intended behavior of CrossView Pro when acting as a COM server; CrossView Pro pauses for the 10 seconds to wait for clients such as MATLAB to release their COM references.

PIL Metrics

The following metrics provide verification information to be used in conjunction with the main “signal level” cosimulation results:

- C Code Coverage reports
- Execution profiling
- Stack profiling

C Code Coverage Reports

After you download a PIL application and run a cosimulation, you can view reports in MATLAB. The reports available depend on the target configuration. For the C166 Simulator, a hyperlink is provided for each report in the MATLAB command window towards the end of the Real-Time Workshop build log (as shown in the following example):

```
PIL reports available from CrossView Pro for block: fuelsys
Coverage ("covinfo"):  Yes (pil_coverage_report)
Profiling ("proinfo"): Yes (pil_profiling_report)
Cumulative profiling ("cproinfo"):  Yes
(pil_cumulative_profiling_report)
```

To view the C code coverage report, click the hyperlink pil_coverage_report:

```
pil_coverage_report =

Module:   temp                                     0%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim_pil...
\pil_interface.c    74%
Function: pilInitialize                           75%
Function: pilGetUDataSymbol                       75%
Function: pilStep                                 71%
Function: pilGetYDataSymbol                       75%
Function: pilTerminate                            75%
Module:   ..\..\..\..\..\matlab\toolbox\rtw\targets\common...
\tgtcommon\pilsrc\pil_ide_data_stream.c    93%
Function: pilDataBreakpoint                      100%
Function: pilReadData                             90%
Function: pilWriteData                            94%
Function: pilDataInit                            100%
Module:   ..\..\..\..\..\matlab\toolbox\rtw\targets\common...
\tgtcommon\pilsrc\pil_interface_lib.c    97%
Function: getNextSymbol                          100%
Function: processData                             93%
Function: pilCommandLoop                          99%
Module:   ..\..\..\..\..\matlab\toolbox\rtw\targets\common...
\tgtcommon\pilsrc\pil_main.c    83%
Function: main                                    83%
```

```
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\fuelsys1.c
    61%
Function: Sens_Failure_Counter                     13%
Function: Fueling_Mode                             24%
Function: Init_controllogic                       100%
Function: controllogic                             47%
Function: fuelsys1_step                            79%
Function: fuelsys1_initialize                     100%
Function: fuelsys1_terminate                      100%
Module:   MEMCPY_C                                100%
Module:   MEMSET_C                                100%
Module:   MUL                                     100%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\binarysearch_s16.c    89%
Function: BINARYSEARCH_S16                         89%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\dotproduct_s32s16.c   100%
Function: DotProduct_s32s16                       100%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\interpolate_even_s16_s16_sat.c    84%
Function: INTERPOLATE_EVEN_S16_S16_SAT             84%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\interpolate_s16_s16_sat.c    83%
Function: INTERPOLATE_S16_S16_SAT                  83%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\look2d_s16_s16_s16_sat.c   100%
Function: Look2D_S16_S16_S16_SAT                  100%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\div_s32_sat_floor.c    77%
Function: div_s32_sat_floor                        77%
Module:   UDIL                                     29%
Module:   UMOL                                     24%
Module:   fuelsys1_pil                              0%
Module:   CSTART                                    0%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\fuelsys1_data.c
     0%
```

Execution Profiling

In this section...

"CrossView Pro Execution Profiling" on page 3-15

"Task Execution Profiling Kit for Real-Time Workshop Targets" on page 3-18

CrossView Pro Execution Profiling

After you download a PIL application and run a cosimulation, you can view reports in MATLAB. The reports available depend on the target configuration. For the C166 Simulator, a hyperlink is provided for each report in the MATLAB command window towards the end of the Real-Time Workshop build log (as shown in the following example):

```
PIL reports available from CrossView Pro for block: fuelsys
Coverage ("covinfo"):  Yes (pil_coverage_report)
Profiling ("proinfo"): Yes (pil_profiling_report)
Cumulative profiling ("cproinfo"):  Yes
(pil_cumulative_profiling_report)

Maximum stack usage during PIL (including the PIL test
framework overhead):

C166 User Stack: 59/109 (54.13%) words used.
```

To view the profiling report, click the hyperlink `pil_profiling_report`:

```
pil_profiling_report =

Total Execution Time:  4447016
                                                Cycles %Cycles
Function: pilInitialize                             16 0.000%
Function: pilGetUDataSymbol                      22428 0.504%
Function: pilStep                                20826 0.468%
Function: pilGetYDataSymbol                      22428 0.504%
Function: pilTerminate                              16 0.000%
Function: pilDataBreakpoint                      14454 0.325%
Function: pilReadData                           549878 12.37%
```

```
Function: pilWriteData                          166816 3.751%
Function: pilDataInit                                4 0.000%
Function: getNextSymbol                          80100 1.801%
Function: processData                           288360 6.484%
Function: pilCommandLoop                        137966 3.102%
Function: main                                    6432 0.145%
Function: Sens_Failure_Counter                   22400 0.504%
Function: Fueling_Mode                           54740 1.231%
Function: Init_controllogic                         58 0.001%
Function: controllogic                          121744 2.738%
Function: fuelsys1_step                         677674 15.24%
Function: fuelsys1_initialize                       48 0.001%
Function: fuelsys1_terminate                         4 0.000%
Function: BINARYSEARCH_S16                      372458 8.375%
Function: DotProduct_s32s16                      37642 0.846%
Function: INTERPOLATE_EVEN_S16_S16_SAT           51678 1.162%
Function: INTERPOLATE_S16_S16_SAT               528118 11.88%
Function: Look2D_S16_S16_S16_SAT                256320 5.764%
Function: div_s32_sat_floor                     406596 9.143%
Module:   temp                                       0 0.000%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim_pil\pil_interface.c
    65714 1.478%
Module:   ..\..\..\..\..\sandbox\targets with spaces\matlab...
\toolbox\rtw\targets\common\tgtcommon\pilsrc\...
pil_ide_data_stream.c   731152 16.44%
Module:   ..\..\..\..\..\sandbox\targets with spaces\matlab...
\toolbox\rtw\targets\common\tgtcommon\pilsrc\...
pil_interface_lib.c   506426 11.39%
Module:   ..\..\..\..\..\sandbox\targets with spaces\matlab...
\toolbox\rtw\targets\common\tgtcommon\...
pilsrc\pil_main.c     6432 0.145%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\...
fuelsys1.c   876668 19.71%
Module:   MEMCPY_C                              357522 8.040%
Module:   MEMSET_C                                 792 0.018%
Module:   MUL                                    13398 0.301%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
binarysearch_s16.c   372458 8.375%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
dotproduct_s32s16.c    37642 0.846%
```

```
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
interpolate_even_s16_s16_sat.c    51678 1.162%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
interpolate_s16_s16_sat.c   528118 11.88%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
look2d_s16_s16_s16_sat.c   256320 5.764%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
div_s32_sat_floor.c   406596 9.143%
Module:   UDIL                                  147648 3.320%
Module:   UMOL                                   85064 1.913%
Module:   fuelsys1_pil                               0 0.000%
Module:   CSTART                                     0 0.000%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\...
fuelsys1_data.c        0 0.000%
27:     readDataPtr = & pil_ide_data_buffer[0];
```

For cumulative profiling, command line messages like the following inform you that you must configure CrossView Pro to specify which functions to collect data for. Select **Tools > Cumulative Profiling Setup**, specify functions, and then run the cosimulation again to get the report.

```
NOTE: Cumulative profiling requires manual setup in
CrossView Pro.
See Tools->Cumulative Profiling Setup
DO NOT add the function pilDataBreakpoint to the list of
functions to profile.

You must then run the PIL simulation again
to generate the report.

pil_cumulative_profiling_report =

CrossView Cumulative Profiling Report
-------------------------------------
Total Execution Time:  4447016
Function                            Calls    Recursive
Min.Time   Max.Time   Avg.Time   Total Time %Time
```

For information on build messages containing links at the command line, see "Command Line Project Information" on page 2-10.

Task Execution Profiling Kit for Real-Time Workshop Targets

This kit, available on MATLAB Central, provides instructions and examples on how to implement real-time task based execution profiling on a custom target. A graphical representation of on-target execution and a HTML report are provided for analysis. You can implement this for your own custom system target file that uses the Embedded IDE Link project generator.

For details, see http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12731

Stack Profiling

In this section...

"What Is Stack Profiling?" on page 3-19

"PIL Applications" on page 3-19

"Non-PIL Applications" on page 3-20

"Infineon® TriCore Stack Depth Analyzer" on page 3-21

What Is Stack Profiling?

Stack profiling gives you a maximum bound on the stack usage of an application. The stack profiling feature works by first writing a signature to the stack memory region, then when the application executes normally, the signature pattern is overwritten by the application stack data. Finally the stack memory is read into MATLAB and analyzed to determine how much of the stack memory was used during execution.

PIL Applications

Stack profiling is automatically reported after PIL cosimulation. The report gives you a maximum bound on the stack usage of the algorithm under test.

Output at end of PIL (bold indicates hyperlinks):

```
Maximum stack usage during PIL (including the PIL
test framework overhead):
TriCore User Stack: 24/2048 (1%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.
```

The hyperlinks for the individual stacks expand to more information about that stack, as shown in the following example.

```
PIL reports available from CrossView Pro for block: fuelsys

Coverage ("covinfo"):             No
Profiling ("proinfo"):            Yes (pil_profiling_report)
Cumulative profiling ("cproinfo"): Yes (pil_cumulative_profiling_report)

Maximum stack usage during PIL (including the PIL test framework overhead):

TriCore User Stack: 24/2048 (1%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.

         name: TriCore User Stack
  baseAddress: 0xA0080130 (2684879152 decimal)
   endAddress: 0xA008212F (2684887343 decimal)
      memSize: 0x2000 (8192 decimal) memory units
growDirection: down
  memorySpace: N/A
```

The hyperlink for "including the PIL test framework overhead" expands to show this explanation:

PIL Test Framework Overhead: The maximum stack usage reported after PIL is the stack usage of the entire PIL application, which includes a small amount of stack used by the PIL test framework. The stack usage reported is therefore a maximum bound on the stack usage of the algorithm under test.

To more accurately determine the stack usage of the algorithm it is possible to use the Embedded IDE Link CrossView Pro stack profiling feature on an application that is not configured for PIL. This will allow the stack usage to be determined without the stack overhead of the PIL test framework.

Non-PIL Applications

Non-PIL applications (perhaps with stimulus signals coming from target I/O drivers) can be profiled using the CrossView Pro API commands `stackProfileReset` and `stackProfile`.

1. Call `stackProfileReset` to reset the application you are debugging, and write a signature pattern to the stack memory region. Use the following syntax:

   ```
   xview.stackProfileReset
   ```

   where *`xview`* is a `tasking.xviewapi` object. See "Methods for Class tasking.xviewapi" on page 2-24.

2 Call `stackProfile` immediately after resetting to view 0% stack usage profiling results.

3 Execute the application (e.g., `xview.execute('C')`).

4 After the amount of time you want to profile for, stop the application using `xview.halt`

5 Call `stackProfile` to get the profiling results for the execution period.

An example of this procedure is shown following.

```
>> XView_Obj.stackProfileReset
CrossView Pro: A hardware reset occurred.
CrossView Pro: A software reset of the program occurred.
>> XView_Obj.stackProfile
Maximum stack usage since last profile reset:

TriCore User Stack: 0/2048 (0%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.

>> XView_Obj.execute('C');
CrossView Pro: The target is running.
>> XView_Obj.halt
CrossView Pro: The target stopped executing with cause: "UNKNOWN"
CrossView Pro: Command with sequence number 71 completed.
>> XView_Obj.stackProfile
Maximum stack usage since last profile reset:

TriCore User Stack: 4/2048 (0%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.
```

Infineon TriCore Stack Depth Analyzer

The Infineon TriCoreStack Depth Analyzer (SDA) tool is a static stack depth analyzer for the TASKING TriCore toolset. This is an alternative to the dynamic stack profiling provided with Embedded IDE Link software.

It can be found at the Infineon TriCore Software Downloads page. Navigate there from this URL:

```
http://www.infineon.com/tricore
```

Click the link on the right: “Development Tools, Software and Training”, then click “Software Downloads”.

Bidirectional Traceability Between Code and Model

In this section...
"Using Traceability" on page 3-22
"Enabling Traceability" on page 3-23

Using Traceability

Context menu items and command-line methods allow you to navigate bidirectionally between Simulink blocks and the corresponding generated source files in the TASKING EDE or the CrossView Pro debugger.

See the demo tasking_demo_objects to try this feature. This is a command line demo that you can run from the Help browser.

To find the generated code for any block in the model, right click on the block and select: **Embedded IDE Link > See Generated Code in EDE** or **Embedded IDE Link > See Generated Code in CrossView Pro**.

This opens the source file which contains the generated code for the block, and highlights the Real-Time Workshop tag for that block. The Real-Time Workshop tag is usually found in the block's generated comments preceding the block's code.

There are command-line alternatives to the right-click context menu items — see `tasking_demo_objects` for an example.

To find the block which corresponds to some generated code in the EDE or CrossView Pro:

1. Click to place the cursor at the line of code containing the Real-Time Workshop Tag for the given block. Here is an example:

   ```
   /* Outputs for atomic SubSystem: '<Root>/SS2'
   ```

2. Enter at the MATLAB command prompt:

   ```
   EDE_Obj.hilite_system
   ```

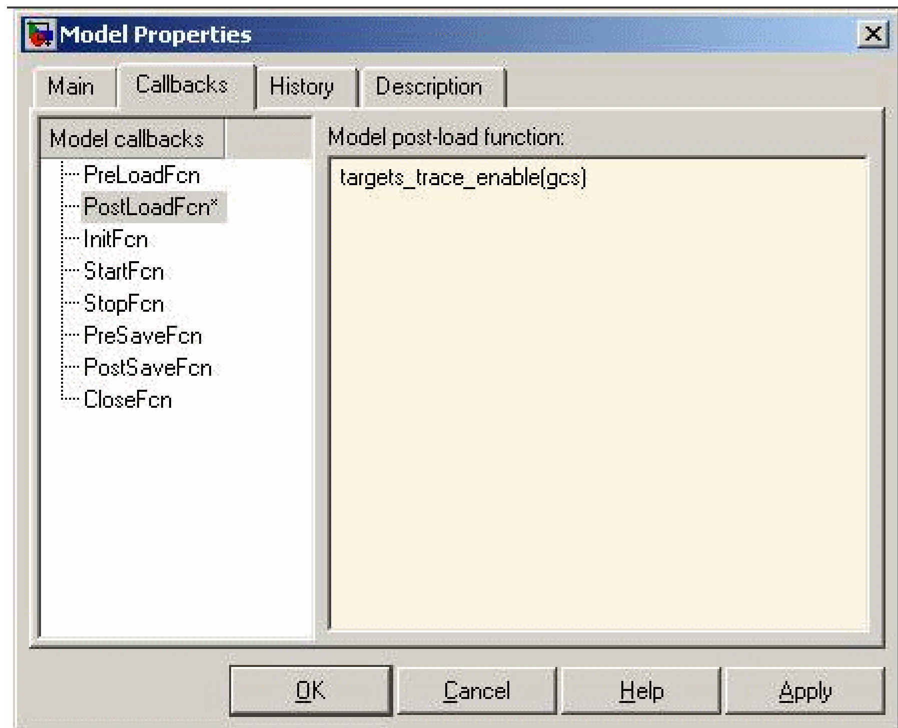
or

```
XView_Obj.hilite_system
```

Enabling Traceability

To use the Traceability feature, you must configure your model as follows:

1. Enable the generation of traceability information by adding `targets_trace_enable(gcs)` to the `PostLoadFcn` callback of the model.
   a. Select **File > Model Properties > Callbacks**.'
   b. Click `PostLoadFcn`.
   c. Enter `targets_trace_enable(gcs)`, as shown.



Click **OK**.

Note `targets_trace_enable` also selects the check box options **Create Code Generation report** and **Code-to-model** under **Report** in the Configuration Parameters dialog box.

---

2 The model must use an ERT based Target.

MISRA C Rule Checking

The TASKING C compiler supports MISRA C rule checking and can be easily configured to check the code generated by Real-Time Workshop software.

You can switch on MISRA C rule checking in your application and/or library template projects. When you build using these template projects, the TASKING compiler will provide warnings about MISRA C violations. Embedded IDE Link software returns these warnings to the MATLAB command line for your review.

Embedded IDE Link software provides an example application project template, pre-configured for MISRA C rule checking, for the TASKING TriCore Toolset. For instructions, see the MISRA C Rule Checking demo, `tasking_demo_misra.m`.

4

Optimization

- "Compiler / Linker Optimization Settings" on page 4-2
- "Target Memory Placement / Mapping" on page 4-3
- "Execution and Stack Profiling" on page 4-4
- "Target Specific Optimizations" on page 4-5
- "Model Advisor" on page 4-9

Compiler / Linker Optimization Settings

Template projects allow you to fully control the optimization settings used by the compiler and linker.

- See "Template Projects" on page 2-4 for details of using template projects.
- See "PIL Block Parameters" on page 3-9 for information about optimization setting requirements for Processor-in-the-Loop.
- See the TASKING documentation for details of available optimization settings.

Target Memory Placement / Mapping

Template projects allow you to fully control the target memory map used for your application.

- See "Overview of the Project Generator Component" on page 2-2 for a general discussion of how the code generation process and subsequent build process work together, including a memory placement example.
- See "Template Projects" on page 2-4 for details of using template projects. See the TASKING documentation for details of memory map settings.

Execution and Stack Profiling

In this section...
"Execution Profiling" on page 4-4
"Stack Profiling" on page 4-4

Execution Profiling

Execution profiling metrics from the CrossView Pro instruction set simulator during PIL cosimulation can be used to identify areas of your algorithms that can be further optimized.

See "Execution Profiling" on page 3-15 for details.

Stack Profiling

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

See "Stack Profiling" on page 3-19 for details.

Target Specific Optimizations

In this section...
"C Language Extensions / Intrinsics" on page 4-5
"Target Optimized Libraries for Infineon XC166 and Infineon® TriCore" on page 4-7
"Target Optimized FIR / FFT Blocks for the Infineon® TriCore" on page 4-8

C Language Extensions / Intrinsics

Infineon TriCore

Support	C89/C90 ANSI Target Function Library	Infineon TriCore ISO Target Function Library	Infineon TriCore Target Function Library (ERT Only)
ANSI Support	Yes	Yes	Yes
ISO Support		Yes	Yes
Saturated Arithmetic Support			Yes

ISO/IEC 9899:1999 Math Library. The target function library `Infineon TriCore ISO`uses the TASKING ISO/IEC 9899:1999 Math Library to implement floating-point mathematical function blocks (e.g. trigonometric functions, log functions). Using these target optimizations improves the performance of applications performing floating-point mathematical operations.

When using these target optimizations, the regular Real-Time Workshop implementation for many ANSI floating-point mathematical operations is replaced by the ISO equivalent. These functions behave identically to the regular Real-Time Workshop implementation and can be verified using processor-in-the-loop cosimulation.

You can use the `Infineon TriCore ISO` target function library with ERT or GRT system target files.

To enable the math library for the optimization of floating-point mathematical operations, select `Infineon TriCore ISO` for the Real-Time Workshop option **Target function library** (on the **Interface** pane of the Configuration Parameters dialog box).

Saturated Arithmetic. The target function library `Infineon TriCore` includes all ISO optimizations and also saturated arithmetic optimizations. The target function library `Infineon TriCore` is only available for ERT system target files.

You can use TASKING compiler extensions and intrinsic functions for saturated arithmetic. These target optimizations can increase execution speed up to 18 times for saturated arithmetic operations. The use of these target optimizations will improve the performance of most applications performing saturated arithmetic operations. It is therefore recommended to enable the optimizations.

When using these target optimizations, the regular Real-Time Workshop implementation for many saturated arithmetic operations are replaced by calls to target optimized inlined functions. The behavior of these functions is identical to the regular Real-Time Workshop implementation and can be verified using processor-in-the-loop cosimulation (see "Processor-in-the-Loop (PIL) Cosimulation" on page 3-2).

To enable TASKING compiler extensions and intrinsic functions for the optimization of saturated arithmetic, select `Infineon TriCore` for the Real-Time Workshop option **Target function library** (on the **Interface** pane of the Configuration Parameters dialog box).

General

Depending on your toolset, your the TASKING compiler may support C language extensions or intrinsics to help optimize in some of the following areas:

- Data Types (eg. Fractional Arithmetic, Bit Addressable Memory)
- Memory Qualifiers (eg. Near, far address space)

- Data Type Qualifiers (eg. Circular Buffers, Saturated arithmetic)

Please see your TASKING documentation for details. You can use these language extensions in your own Simulink blocks and / or custom code.

Target Optimized Libraries for Infineon XC166 and Infineon TriCore

The following optimized libraries are available for the processors supported by Embedded IDE Link software, and can be used to create optimized Simulink blocks:

- Infineon XC166 DSP Library for TASKING compiler

  This library is described by Infineon as follows:

  XC166 DSP library is a DSP function library, is C-callable, manually coded assembly, general purpose signal processing routines:

  - Arithmetic Functions
  - Filters (FIR–, IIR–, Adaptive Filters)
  - Transforms (FFT, IFFT)
  - Matrix Operations
  - Mathematical Operations
  - Statistical Functions

  See the Infineon C166 Software Downloads Web page to get the XC166 DSP Library. Navigate there from this URL:

  ```
  http://www.infineon.com/c166
  ```

  Click the link on the right: "Development Tools, Software and Training", then click "Software Downloads".

- Infineon TriCore DSP Library (TriLib)

  This library is described by Infineon as follows:

  TriLib is a DSP Library for TriCore, containing more than 60 commonly used DSP routines for

  - Complex & Vector Arithmetic

- FIR, IIR, Adaptive Filters
- Fast Fourier, Discrete Cosine Transform
- Mathematical, Matrix, Statistical functions

See the Infineon TriCore Software Downloads page to get the TriLib DSP Library. Navigate there from this URL:

```
http://www.infineon.com/tricore
```

Click the link on the right: "Development Tools, Software and Training", then click "Software Downloads".

Target Optimized FIR / FFT Blocks for the Infineon TriCore

Example FIR / FFT blocks that call target optimized Infineon TriLib routines are available on MATLAB Central. These blocks can be over a hundred times faster than the regular blocks in the Signal Processing Blockset product.

Search MATLAB Central for details.

Model Advisor

Following the suggestions in the Model Advisor report may result in faster on-target execution. See “Consulting the Model Advisor” in the Simulink documentation.

Tutorials

- "Tutorial: Using Option Sets" on page 5-2
- "Tutorial: Creating New Template Projects" on page 5-4
- "Tutorial: Configuring an Existing Model for Embedded IDE Link Software" on page 5-9

Tutorial: Using Option Sets

Option sets are preconfigured settings to specify the target configuration for the TASKING tools. You use option sets to apply EDE project settings (e.g., compiler and linker settings, hardware or simulator) that you can then modify if you choose. For example, once you have set up your target preferences for a TriCore configuration, you can use option sets to switch between using an instruction set simulator configuration, two hardware board configurations, or a simulator with some MISRA C rule checking.

To choose an option set:

1. Enter `taskingutils` in the Command Window.

   The Embedded IDE Link Utilities for Use with Tasking dialog box appears.

2. Select `Target Preferences` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

3. Select a target configuration (e.g., `C166`, `TriCore`) from the list in the dialog box, and click **OK**.

   The Option Set Selection dialog box appears.

4. Select an option set. The list items are specific to the configuration you selected; the available option sets are listed in “Option Sets” on page 1-30. Click **OK**.

   Your target preferences are automatically updated according to the option set you select, and command line messages inform you the following target preferences have changed:

   - `EDE_Configuration`

     `Template_Application_Project`: Set to default template application project relating to the option set.

     `Template_Library_Project`: Set to default template library project relating to the option set.

   - `CrossView_Pro_Configuration`

`Initialization_File`: Set to CrossView Pro (`.st`) initialization file relating to the option set.

Now, when you build any model configured for the same target (e.g., `TriCore`), these project settings are used. To switch to a different option set, repeat the steps.

You can also use option sets to set up an initial configuration when creating new template projects. See "Tutorial: Creating New Template Projects" on page 5-4.

Tutorial: Creating New Template Projects

In this section...
"Creating New Template Projects" on page 5-4
"Creating a New Configuration" on page 5-7

Creating New Template Projects

In this tutorial, you create new template projects for a target configuration and set up options such as simulator or hardware implementation, compiler and linker settings, MISRA C rule checking, or any other project options. Every time you build a model for the selected target configuration, the project options you have set up in the new template projects are used.

Note You may want to create a new configuration to use with new template projects. See the next section, "Creating a New Configuration" on page 5-7 for details.

To create custom application and library template projects:

1. Enter `taskingutils` in the Command Window.

   The Embedded IDE Link Utilities for Use with Tasking dialog box appears.

2. Select `Create New Template Projects` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

3. Select your target (e.g., `TriCore`), and click **OK**.

   Your target preferences for the location of your TASKING installation must be set up for the target configuration you choose (see "Setting Target Preferences" on page 1-10).

   a. Make sure the fields are filled in for this configuration (except the Application and Library Template Projects fields, and CrossView Initialization field, which are autopopulated during the following steps).

b If your target preferences are set up correctly, your TASKING EDE launches when you click **OK**.

4 When you are prompted, choose a location for the template projects, and enter the template name.

5 When you are prompted, choose an option set. An *option set* delineates options specific to your target, such as whether you want to use the simulator or hardware. You can use these to set up an initial configuration to modify later. See "Option Sets" on page 1-30 for more information and a list of available option sets.

You now have custom template projects for this new configuration. The EDE project settings associated with the option set are applied to the new template projects. Messages at the command line inform you the following target preferences have been automatically updated:

- EDE_Configuration

  `Template_Application_Project`: Set to new template application project configured by the option set.

  `Template_Library_Project`: Set to new template library project configured by the option set.

- CrossView_Pro_Configuration

  `Initialization_File`: Set to CrossView Pro (`.st`) initialization file configured by the option set.

Note You can always choose a preconfigured option set (from the drop-down list of the **Configuration** field) to return to the default settings.

Next, modify the compiler settings for these new template projects.

6 To modify the template projects, you need to open them in the TASKING EDE:

a Enter `taskingutils` in the Command Window.

The Embedded IDE Link Utilities for Use with Tasking dialog box appears.

b Select `Open Existing Template Projects` from the list in the dialog box, and click **OK**.

The Target Preferences Configuration Selection dialog box appears.

c Select the same target for which you created new template projects, and click **OK**.

The template projects should now be open in the EDE.

Note Opening or making changes to template projects causes the regeneration of application and library projects.

---

d Right-click the project in the TASKING EDE, and select **Project Options**. You can now modify the project options (compiler settings, linker settings, etc.).

Note When making any changes to template projects, it is important to remove the project from the project space, to make sure your changes are written to disk. Otherwise the changes may not be applied immediately. To remove a current project from the project space, right-click on it and choose **Remove from Project Space**.

---

e When done, close the template projects in the TASKING EDE.

7 To modify your CrossView Pro configuration (optional) you need to specify a `.ini` file in the `Initialization_File` Target Preference field. See `Initialization` in the section "Target Preference Fields" on page 1-13.

You are now ready to use the configuration.

8 Open any Simulink model that is configured with the Embedded IDE Link component (`tasking_demo_fuelsys`, for example).

9. Select **Simulation > Configuration Parameters**. The Configuration Parameters dialog box opens.

10. Select **Embedded IDE Link** on the left-side panel. When you select your target in the **Target Preference Configuration** menu, the template projects you have set up are used.

See "Template Projects" on page 2-4 for details about how Embedded IDE Link software uses template projects during the build process.

Creating a New Configuration

You can customize the default Target Preference configurations by choosing from the preconfigured options sets, or by creating new template projects.

However, it may be useful to create a new Target Preference configuration if you want to switch between them in the **Target Preference Configuration** menu. For example, if your target is a TriCore processor, you could set up a new configuration called `TriCore_user` to specify hardware settings for your target; then you can easily switch between `TriCore` (the default instruction set simulator configuration) and `TriCore_user` using the **Target Preference Configuration** menu in your model's Configuration Parameters dialog box.

In this tutorial, you create a new TASKING configuration and save it in the target preferences. You can then use your new configuration in any Simulink model that is configured with Embedded IDE Link software by selecting it in the **Target Preference Configuration** menu.

To create a new configuration:

1. Enter `taskingutils` in the Command Window.

   The Embedded IDE Link Utilities for Use with Tasking dialog box appears.

2. Select `Target Preferences` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

3. Select `Create new Configuration`, and click **OK**.

4. Expand `Configuration_Options`.

5 Type `Tutorial` in the **Configuration_Description** field.

6 Fill in the rest of the fields for this configuration. See "Setting Target Preferences" on page 1-10 to set these fields properly.

   a You must specify the location of your toolset, by filling in the path to the `CrossView_Pro_Executable`, the `DOL_File`, and the `EDE_Executable`.

   b You can set up the template projects and CrossView initialization fields automatically in one of two ways:

      - You can use the **Start** menu option **Select Preconfigured Target Preference Settings**. See "Tutorial: Using Option Sets" on page 5-2 for instructions.

      - You can create new template projects for this configuration. See "Tutorial: Creating New Template Projects" on page 5-4.

   If you are going to use either of these options you can leave the template projects and CrossView initialization fields blank, because they will be filled in automatically when you follow the steps in using option sets or creating new template projects.

   Click **OK** to close and save your target preferences.

7 After you save your target preferences, you can use the new `Tutorial` configuration in any model that is configured with Embedded IDE Link software. For example, open any of the Embedded IDE Link demo models (such as `tasking_demo_fuelsys`).

8 Select **Simulation > Configuration Parameters**. The Configuration Parameters dialog box opens.

9 Select **Embedded IDE Link** on the left-side panel. Click the **Target Preference Configuration** menu, and notice that the `Tutorial` configuration now appears in the list.

Tutorial: Configuring an Existing Model for Embedded IDE Link Software

In this tutorial, you configure an existing fixed-point model and build it with Embedded IDE Link software.

1. At the MATLAB command prompt, type `rtwdemo_fixptdiv` to open a fixed-point demo model.

2. Switch the model to use Real-Time Workshop Embedded Coder software as follows:

   a. Select **Simulation > Configuration Parameters**, and click **Real-Time Workshop**.

   b. Click **Browse** and select `ert.tlc` (first item in the list). Click **OK**.

3. Select **Tools > Embedded IDE Link > Add Embedded IDE Link Configuration to Model** to add the Embedded IDE Link configuration set to the model.

4. Open the Configuration Parameters dialog box again from the **Simulation** menu, and verify that the Embedded IDE Link configuration set is now added to the model. Select **Embedded IDE Link** from the left panel:

   a. Set the **Build Action** to `Create and Build Application Project`.

   b. Select the **Target Preference Configuration** to match your target.

   c. Select the check box option to **Add Build Directory Suffix**, and type `int` in the **Build Directory Suffix** field.

   d. Under the **Real-Time Workshop** options, select **Interface** and clear the check box for **floating-point numbers** support under **Software environment**, because this model is fixed point. Clearing this option instructs Real-Time Workshop software to avoid building the floating-point version of the `rtwlib` library.

   e. Under **Real-Time Workshop**, select **Hardware Implementation**, and select your device type. For example:

      - For C166 platforms, select `Infineon C16x, XC16x`.
      - For TriCore platforms, select `Infineon TriCore`.

- For ARM platforms, select `ARM 7/8/9`.
- For Renesas M16C, 8051 Compatible, or Freescale DSP563xx (16-bit mode) platforms, select those options.

You are now ready to build the model. Press **Ctrl+B** or select **Tools > Real-Time Workshop > Build Model**.

Configuration Parameters

Embedded IDE Link Pane

Embedded IDE Link

Build Configuration

Build action: Create Application Project

Target Preferences Configuration: C166 Edit Configuration

Add build directory suffix

Build directory suffix: fp

Export Handles

Export EDE handle to MATLAB base workspace

EDE handle name: EDE_Obj

Export CrossView Pro handle to MATLAB base workspace

CrossView Pro handle name: XView_Obj

Processor-in-the-Loop (PIL) Verification

Configure model to build PIL algorithm object code

PIL block action: None

In this section...

"Overview" on page 6-3
"Build Action" on page 6-4
"Target Preference Configuration" on page 6-6
"Add build directory suffix" on page 6-7
"Build directory suffix" on page 6-8
"Export EDE handle to MATLAB base workspace" on page 6-9
"EDE handle name" on page 6-9
"Export CrossView Pro handle to MATLAB base workspace" on page 6-11
"CrossView Pro handle name" on page 6-11
"Configure model to build PIL algorithm object code" on page 6-13
"PIL block action" on page 6-14

Overview

Parameters for controlling Embedded IDE Link build configuration, export handles, and processor-in-the-loop verification.

Configuration

This pane appears if you add the Embedded IDE Link configuration options to a model with any system target file. To do this, select the menu item **Tools > Embedded IDE Link > Add Embedded IDE Link Configuration to Model**.

See Also

Working with Configuration Sets

Build Action

Set what action to take after the Real-Time Workshop build process completes. You can create application and library projects in the TASKING EDE and then stop, or you can also choose to build, execute, or debug.

Settings

Default: `Create Application Project`

`Create Application Project`
: Generate code for the model or subsystem, create a TASKING application project for the selected TASKING configuration, connect to the TASKING EDE, and open the application project (in addition to the required Real-Time Workshop and Signal Processing Blockset Library projects, if required) in the TASKING EDE. This option does not build or execute the application.

`Create Library Project`
: Performs the same actions as Create Application Project, but this option archives the generated code into a library in TASKING. No `main.c` file is generated.

`Create and Build Application Project`
: Performs the same actions as Create Application Project, but also instructs TASKING to build the application project.

`Create and Build Library Project`
: Performs the same actions as Create Library Project, but also instructs TASKING to build the Library project.

`Create, Build and Execute Application Project`
: Performs the same actions as Create and Build Application Project and also downloads the executable file to your CrossView Target and runs the executable. No debugging information is downloaded into the target with this option.

`Create, Build and Debug Application Project`
: Performs the same actions as Create, Build and Execute Application Project but also downloads debugging information to the target. This option behaves the same way as the Debug Application icon in the TASKING EDE.

Tip

To manually debug the executable from the application project, use the Create and Build Application Project option, then click the Debug Application icon in the TASKING EDE

Dependency

This parameter is disabled by **Configure model to build PIL algorithm object code**.

Command-Line Information

Parameter: `TaskingBuildAction`
Type: string
Value: `'Create Application Project'` | `'Create Library Project'` | `'Create and Build Application Project'` | `'Create and Build Library Project'` | `'Create, Build and Execute Application Project'` | `'Create, Build and Debug Application Project'`
Default: `'Create Application Project'`

Recommended Settings

Application	Setting
Debugging	'Create, Build and Debug Application Project'
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Setting Build Action

Target Preference Configuration

Select a configuration description, as defined in the Target Preferences, to be used by the build action.

Settings

Default: `'Target Preference Configuration Not Set'`

After you have set up target preferences for particular configurations, you can select them here (e.g., `'c166'`). The names in the list correspond to the Configuration Description for each configuration in the Embedded IDE Link Target Preferences dialog box. Click **Edit Configuration** to open the Embedded IDE Link Target Preferences dialog box for the currently selected configuration. For instructions, see Using Configuration Sets to Specify Your Target.

Command-Line Information

Parameter: `TaskingConfiguration`
Type: string
Value: `'Target Preference Configuration Not Set'` | Any "Configuration_Description" name defined in the Embedded IDE Link Target Preferences (e.g. `'TriCore'`, `'C166'`, etc.)
Default: `'Target Preference Configuration Not Set'`

See Also

- Using Configuration Sets to Specify Your Target
- Setting Target Preferences

Add build directory suffix

Specify whether to add a model-specific suffix to the regular Real-Time Workshop build folder suffix.

Settings

Default: Off

On

Specify a model-specific suffix to be added the regular Real-Time Workshop build folder suffix. This setting is useful to avoid "shared utility function" code generation errors which occur because of conflicts over Real-Time Workshop utility functions shared between different models. A typical conflict is between with models with floating-point number support and those without. To resolve this conflict, you can add an 'fp' suffix for floating-point models, and an 'int' suffix for non-floating-point models.

Off

Use the default Real-Time Workshop build folder suffix — not using an additional suffix may result in rebuilding shared libraries unnecessarily.

Dependencies

This parameter enables **Build directory suffix**.

Command-Line Information

Parameter: `TaskingSpecifyBuildSubDirName`
Type: logical
Value: `0 | 1`
Default: `0`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	On
Safety precaution	No impact

See Also

Shared Libraries

Build directory suffix

Specify a model-specific suffix to be added the regular Real-Time Workshop build folder suffix.

Settings

No Default

Enter a model-specific suffix to be added the build folder name. This setting is useful to avoid "shared utility function" code generation errors which occur because of conflicts over Real-Time Workshop utility functions shared between different models.

Dependencies

This parameter is enabled by **Add build directory suffix**.

Command-Line Information

Parameter: `TaskingBuildSubDirName`
Type: string
Value: Any string composed of the following characters: [a-z_A-Z0-9]
Default: `''`

See Also

Shared Libraries

Export EDE handle to MATLAB base workspace

Specify whether to export the EDE object handle to the workspace.

Settings

Default: On

On

Export a TASKING EDE object handle to the MATLAB base workspace after the build process completes.

Off

Do not export the EDE object handle to the workspace.

Dependencies

This parameter enables **EDE handle name**.

Command-Line Information

Parameter: `TaskingExportEDEHandle`
Type: logical
Value: `0 | 1`
Default: `1`

See Also

Automation Interface

EDE handle name

Specify a name for the exported handle.

Settings

Default: 'EDE_Obj'

Specify the MATLAB base workspace variable name to export the handle to.

Dependencies

This parameter is enabled by **Export EDE handle to MATLAB base workspace**.

Command-Line Information

Parameter: `TaskingExportEDEHandleName`
Type: string
Value: Any valid MATLAB variable name (see MATLAB function: `isvarname`)
Default: `'EDE_Obj'`

See Also

Automation Interface

Export CrossView Pro handle to MATLAB base workspace

Specify whether to export the CrossView Pro object handle to the workspace.

Settings

Default: On

On

Export the TASKING CrossView Pro object handle to the MATLAB base workspace after the build process completes.

The handle is only exported if the build action launches CrossView Pro.

Off

Do not export the CrossView Pro object handle to the workspace.

Dependencies

This parameter enables **CrossView Pro handle name**.

Command-Line Information

Parameter: `TaskingExportCrossViewHandle`
Type: logical
Value: `0 | 1`
Default: `1`

See Also

Automation Interface

CrossView Pro handle name

Specify a name for the exported handle.

Settings

Default: 'XView_Obj'

Specify the MATLAB base workspace variable name to export the handle to.

Dependency

This parameter is enabled by **Export CrossView Pro handle to MATLAB base workspace**.

Command-Line Information

Parameter: `TaskingExportCrossViewHandleName`
Type: string
Value: Any valid MATLAB variable name (see MATLAB function: `isvarname`)
Default: `'XView_Obj'`

See Also

Automation Interface

Configure model to build PIL algorithm object code

Specify whether to build Processor-in-the-Loop (PIL) algorithm code.

Settings

Default: Off

On

Configure the model to build PIL algorithm code that is suitable for use with the PIL block.

Off

Do not build PIL algorithm code.

Dependency

This parameter enables **PIL block action**.

This parameter disables **Build action**.

See Also

Processor-in-the-Loop (PIL) Cosimulation

PIL block action

Select a PIL block action to take after the Real-Time Workshop build process completes

Settings

Default: `'None'`

`'None'`
: Do not create a PIL block. Choose this to avoid creating a PIL block, for instance if you have already built a PIL block and do not want to repeat the action.

`'Create PIL block, then build and download PIL application'`
: Create the PIL block, then automatically build and download the PIL application. This is the default when you select the option to configure the model for PIL.

`'Create PIL block'`
: Create the PIL block, and then stop without building. You can build manually from the PIL block.

Dependency

This parameter is enabled by **Configure model to build PIL algorithm object code**. When enabled, the default changes to `'Create PIL block, then build and download PIL application'`.

Command-Line Information

Parameter: `TaskingPILBlockAction`
Type: string
Value: `'None'` | `'Create PIL block'` | `'Create PIL block, then build and download PIL application'`
Default: `'None'`

See Also

Processor-in-the-Loop (PIL) Cosimulation

Limitations and Tips

- “General Issues” on page A-2
- “Debugger Issues” on page A-4
- “Build Process Issues” on page A-6
- “Processor-in-the-Loop Issues” on page A-16
- “Issues Using Real-Time Workshop Software Without Real-Time Workshop® Embedded Coder Software” on page A-19

General Issues

In this section...

"Problems with Installations in Read-Only Locations" on page A-2
"Simulink Configuration Set Reference Not Supported" on page A-2
"Serialization of Embedded IDE Link Objects Not Supported" on page A-3

Problems with Installations in Read-Only Locations

The process to build models works correctly when Embedded IDE Link software is installed in a read-only location because the template projects are copied to the working folder during the process. However, installing Embedded IDE Link software in a read-only location (e.g. read-only network) causes the following problems:

- Template project generation fails because the function tasking_generate_templates attempts to write to the installation location.
- Opening existing template projects may fail because the TASKING EDE attempts to write to the installation location.

To resolve this issue:

- Do not install Embedded IDE Link software in a read-only location
- Avoid updating or opening the template projects or temporarily allow write access to the read-only installation location while doing so.
- Create new template projects in a writable location rather than attempting to update the default template projects.

Simulink Configuration Set Reference Not Supported

The Simulink Configuration Set Reference feature is not supported by Embedded IDE Link software.

For Embedded IDE Link software, make sure your model's configuration set objects are "Simulink.ConfigSet" objects and not "Simulink.ConfigSetRef" objects.

Serialization of Embedded IDE Link Objects Not Supported

Serialization (saving and loading to MATLAB .mat file) of the objects provided with Embedded IDE Link software (e.g., tasking.edeapi, tasking.xviewapi) is not possible. If you attempt to load a serialized object from a .mat file you may see the Target Preferences Configuration Selection GUI, warning or error messages, or both.

In some circumstances, a product (for example the SystemTest™ product) or a user script may automatically save all contents of the MATLAB base workspace to a .mat file. In this case, it may be useful to turn off the "Export Handles" settings in the Embedded IDE Link configuration set component. Doing so stops EDE and CrossView Pro objects from being exported to the base workspace at the end of a Embedded IDE Link build process and thus avoids potential serialization problems.

Debugger Issues

In this section...
"ARM CrossView Pro Debugger Fails with File | Open Source Content" on page A-4
"On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware" on page A-4

ARM CrossView Pro Debugger Fails with File | Open Source Content

Due to a CrossView Pro bug, the File | Open Source menu item of the ARM CrossView Pro debugger may fail to open the specified source file. Instead, you may see a blank window or the wrong source file may be opened.

This limitation can affect the Traceability feature from the model to the code in CrossView. If you right-click on a block in the Simulink model and select **Embedded IDE Link > See Code in CrossView Pro**, this operation might not work as expected because the source file cannot be opened.

To work around this issue, you can set a breakpoint in the source file that is initially visible during debugging and step into other source files from there.

On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware

For ARM processors the CrossView Pro instruction set simulator can be used for debugging and processor-in-the-loop (PIL) cosimulation, but there is currently no on-chip debugging or PIL support.

To resolve this problem:

- You can contact TASKING for the latest information on CrossView Pro on-chip debugging support for ARM hardware.
- You can contact Hitex for a solution to debug an application generated by Embedded IDE Link software on ARM hardware, however this solution cannot provide PIL support.

Build Process Issues

In this section...

"Linker Errors Due to Limited Memory" on page A-6

"EDE Is Slow, Unresponsive, or Crashes" on page A-7

"Signal Processing Blockset Library Build Failures" on page A-7

"Memory Block Freed Twice Error" on page A-8

"8051 EDE Cannot Compile Files with Long Names" on page A-8

"DSP563xx Toolset Support Limitations" on page A-8

" "Create, Build and Execute Application Project" Build Action Fails " on page A-9

"C166 Toolset Warnings" on page A-10

"Build Error From Root Drive Location" on page A-10

"Limited Support for Nonfinite Values" on page A-10

"Memory Warning/Error Messages in the CrossView Pro Command Window When Using the Instruction Set Simulator" on page A-13

"C++ Code Generation Not Supported" on page A-13

"Video and Image Processing Blockset Library Not Supported" on page A-14

"Noninlined S-functions Calling rt_matrx.c Not Supported" on page A-14

""Compiler optimization level" Configuration Parameter Has No Effect" on page A-14

"Configuration Changes Cause Build Errors With Referenced Models" on page A-15

Linker Errors Due to Limited Memory

The Embedded IDE Link software supports a variety of targets (instruction set simulators and embedded hardware) with a range of capabilities. Some demo models and user-created models may fail to build for certain targets owing to a lack of available target memory. In such cases you see linker errors like the following:

```
Linking and locating to t_pil_lib_alg_pil.out
E 268: relative linear element 'section T_PIL_LIB_ALG_4_NB class
CNEAR' cannot be located within 4 pages
total errors: 1, warnings: 0
wmk: *** action exited with value 1.
```

To work around such errors you must do one of the following:

1. Modify the model to reduce memory requirements (for example, by optimizing the algorithm, or by using smaller datatypes).
2. Alternatively, modify the target configuration to make more memory available (for example, by using a hardware board with more memory, or changing the memory map to allow extra memory to be used).

In some cases it may not be possible to resolve the problem, because the algorithm represented by the model is too complex for the target.

EDE Is Slow, Unresponsive, or Crashes

Under certain circumstances the TASKING EDE may become slow, unresponsive, or even terminate with virtual memory problems. This limitation is an open issue with the TASKING EDE (for all supported tool suites).

To resolve this issue, take one or both of the following actions:

- Close the EDE and try building the model again
- Try deleting the symbol database file, `cwright.sbl`, which can be found in the EDE_Executable folder ($TASKINGRootDir\bin)

Signal Processing Blockset Library Build Failures

The following problem has been found with Signal Processing Blockset product (“`DSP lib`”) library builds.

With Renesas M16C, building the Signal Processing Blockset library with floating point support enabled results in the following error:

```
TASKING program builder v3.1r1 Build 076 SN 00100552
```

```
Assembling qrdc_z_rt.src asm16c E219:
["qrdc_z_rt.src" 1692] expression out of range
(0 and FF hexadecimal)wmk:
*** action exited with value 1.
```

This limitation is a known issue with the Renesas 16C compiler. To resolve this issue, disable floating point support in the model.

Memory Block Freed Twice Error

Occasionally, when Embedded IDE Link software is creating projects in the TASKING EDE, the following error appears: `Memory block freed twice`. This limitation is a known issue with the TASKING EDE.

To work around the problem, click **OK** in the error dialog box, and the code generation process continues as normal.

8051 EDE Cannot Compile Files with Long Names

If you encounter this problem, you receive an error message similar to the following:

```
Assembling tasking_fuel_controller_ert_rtw_pil_cstart.src
asm51 E001: tasking_fuel_controller_ert_rtw_pil_cstart.src: line 1:
syntax error
wmk: *** action exited with value 1.
```

This message indicates that the full path of the model or subsystem you are trying to build is too long.

To resolve this issue, consider moving the model to a shorter folder name, or renaming the model, subsystem, or both to use shorter names.

DSP563xx Toolset Support Limitations

The following limitations affect use of the DSP563xx Toolset:

- Only 16-bit mode for the DSP563xx Family is supported. Real-Time Workshop `grt.tlc`-based targets and the "GRT Compatible Call interface" option in the **Real-Time Workshop > Interface** settings are not

supported. This limitation is because of the non-standard size of single- and double-precision floating-point datatypes on this architecture (`tmwtypes.h` will not compile)

- The DSP5600x Toolset is NOT supported because none of the processors supported by this toolset have 16-bit memory models.
- Both 16-bit memory models of the DSP563xx Family produce watch errors (wrong values displayed) in CrossView Pro because of an issue with the TASKING toolset. CrossView Pro does not know that the datatype sizes should be different according to the selected memory model. This issue does not affect the DSP566xx Family.

There are no resolutions for this issues.

"Create, Build and Execute Application Project" Build Action Fails

Tool Suites: Renesas M16C

With the Renesas M16C tool suite, if you are executing the application project, rather than debugging (via "Create, Build and Debug Application Project"), this does not work correctly. The application does not execute. This issue occurs because the CrossView Pro Simulator does not know the start address when debugging information is not loaded.

To resolve this issue, perform the following steps after CrossView Pro launches:

1. Stop execution by clicking the Halt button.
2. Execute the following command in the CrossView Pro command window to determine the application entry point stored at location 0xfffffc:

   ```
   *((unsigned long *)0xfffffc)/x
   ```

   Example output for this command is:

   ```
   0xfffffc = 0x000d0000
   ```

3. Change the execution position to the application entry point by executing the "gi" command, using the output of the previous command. For example, 0xd0000 gi

4. Resume execution by clicking the Run/Continue button.

Alternatively, use the “Create, Build and Debug Application Project” build action.

C166 Toolset Warnings

When using the C166 toolset you may see warnings similar to the following:

```
Warning: missing "sdc_lia" or "sdc_lip" lifetime record
```

This warning is caused by a problem with the TASKING toolset and has been registered with Altium as PR35043. It is related to debug life time information.

The warning can be ignored safely.

Build Error From Root Drive Location

On the C166 and 8051 platforms, a limitation of the TASKING toolset may cause build errors if you build from a root drive location such as `c:\` or `d:\`.

Following is an example error with the C166 toolset:

```
cc166: E 014: invalid control:
Files\MATLAB\R2007a_nortwec\toolbox\rtw\targets\c166\c166demos" -Wcp"-IC:\Program
wmk: *** action exited with value 1.
```

Workaround: Always build from a sub-folder location such as `c:\work` or `d:\MATLAB\work`.

Limited Support for Nonfinite Values

Nonfinite values in your model may cause wrong results, linking errors or compilation errors. See below for a possible workaround if your target is a TriCore or ARM platform.

Linking Errors

If you encounter similar linking errors when building your model:

```
undeclared identifier "rtMinusInf"
undeclared identifier "rtNaN"
undeclared identifier "rtInf"
```

then this means that:

- Your model uses nonfinite values, and
- You are using a stubbed version of `rt_nonfinite.c` which does not define `rtMinusInf`, `rtNan`, or the other nonfinite identifiers required by Real-Time Workshop software.

To resolve this issue:

- Do not use nonfinite values in the model. Such values are not desirable for embedded applications. Nonfinite elements on targets other than TriCore or ARM are not supported with Embedded IDE Link software.
- If you want to use nonfinite values and your target is a TriCore or ARM platform, then you can use the following workaround. You do not need to use a stubbed version of `rt_nonfinite.c` because the default one should compile correctly on this 32-bit target. In the configuration set, under **Real-Time Workshop** in **TLC Options**,
  1. Remove `-aCustomNonFinites="genrtnonfinite_stub.tlc"`.
  2. Delete the generated `rt_nonfinite.c` file in the build area before attempting to build the model again. This procedure should generate a new `rt_nonfinite.c` file that correctly defines the undeclared identifiers.

Compilation Errors

If you encounter compilation errors in `rt_nonfinite.c` similar to the following:

```
Compiling and assembling rt_nonfinite.c
..\..\slprj\ert_c167cs_sim\_sharedutils\rt_nonfinite.c:
    47:        uint32_T fraction : 23;
E 134: bitfield size out of range - set to 1
```

```
    57:           uint32_T fraction1 : 20;
E 134: bitfield size out of range - set to 1
    69:       (*(LittleEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
    78:         (*(LittleEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
    89:         (*(LittleEndianIEEEDouble*)&rtNaN).wordL.fraction1 = 0xFFFFF;
W 195: constant expression out of range -- truncated
    90:         (*(LittleEndianIEEEDouble*)&rtNaN).wordH.fraction2 = 0xFFFFFFFF;
W 196: constant expression out of range due to signed/unsigned type mismatch
    98:         uint32_T fraction : 23;
E 134: bitfield size out of range - set to 1
   105:           uint32_T fraction1 : 20;
E 134: bitfield size out of range - set to 1
   118:       (*(BigEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
   127:         (*(BigEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
   138:         (*(BigEndianIEEEDouble*)&rtNaN).wordL.fraction1 = 0xFFFFF;
W 195: constant expression out of range -- truncated
   139:         (*(BigEndianIEEEDouble*)&rtNaN).wordH.fraction2 = 0xFFFFFFFF;
W 196: constant expression out of range due to signed/unsigned type mismatch
total errors: 4, warnings: 8
wmk: *** action exited with value 1.
wmk: *** action exited with value 1.
```

then this issue indicates that you are compiling the default Real-Time Workshop `rt_nonfinite.c` on a target that does not support it. The only targets which can compile the default `rt_nonfinite.c` are the TriCore and ARM platforms. Nonfinite elements on targets other than TriCore or ARM platforms are not supported with Embedded IDE Link software.

To resolve this issue, follow these steps:

1. Make sure you are using the stubbed out version of this file. In the configuration set, under **Real-Time Workshop** in **TLC Options**, add the following: `-aCustomNonFinites="genrtnonfinite_stub.tlc"`

2. Delete the `rt_nonfinite.c` file from the build area before attempting to rebuild the model in the same build area.

Memory Warning/Error Messages in the CrossView Pro Command Window When Using the Instruction Set Simulator

Due to a limitation in the TASKING C166 toolset you may see messages similar to the following in the CrossView Pro command window during execution of an application in the instruction set simulator:

```
GPR registers could not be scheduled to 0xF200
GPR registers could not be scheduled to 0xF220
```

and

```
Reading register "R0" (0) failed: memory failure at
memory space 0 range 0x00FC00-0x00FC01
```

These messages occur because the CrossView Pro feature "Use map file for memory map" does not work correctly.

The workaround suggested by Altium is to not use this feature, in which case the debugger assumes that the entire memory range that the processor can address is available to the application.

You can create custom Embedded IDE Link template projects and a custom CrossView Pro initialization file to disable this feature. For example, in the custom template application project, uncheck the project option, **CrossView Pro > Initialization > Use map file** for memory mapping.

C++ Code Generation Not Supported

C++ code generation is not supported. If you try to use this option, you see an error message like the following:

```
Embedded IDE Link does not support the RTW C++ Target
Language option. Please set the "Language" setting to
"C" in the Real-Time Workshop configuration parameters of
the model.
```

There is no resolution for this issue.

Video and Image Processing Blockset Library Not Supported

The Video and Image Processing Blockset Real-Time Workshop library is not supported by Embedded IDE Link software. If you include blocks from the Video and Image Processing Blockset library in your model then you may see compilation or link errors.

There is no resolution for this issue.

Noninlined S-functions Calling rt_matrx.c Not Supported

Noninlined S-functions that use routines in `rt_matrx.c` are not supported because `rt_matrx.c` contains functions that can allocate memory dynamically. Embedded IDE Link software does not support dynamic memory allocation. You may see errors like the following:

```
Linking and locating to rt_matrx_test.out
E 222: module _nmalloc.obj (_NMALLOC_C):
symbol '?C166_NHEAP_TOP': unresolved
E 222: module _nmalloc.obj (_NMALLOC_C):
symbol '?C166_NHEAP_BOTTOM':
       unresolved
total errors: 2, warnings: 0
```

There is no resolution for this issue.

"Compiler optimization level" Configuration Parameter Has No Effect

When using Embedded IDE Link software, the Real-Time Workshop Configuration Parameter **Compiler optimization level** has no effect on the building of generated code in the TASKING EDE.

The Embedded IDE Link template projects specify the compiler and linker settings used for building the generated code. See "Template Projects" on page 2-4 for more information, and "Tutorial: Creating New Template Projects" on page 5-4 for instructions on customizing settings.

Configuration Changes Cause Build Errors With Referenced Models

If you build a model hierarchy, and then change your option set or template application project before rebuilding, you see build errors like the following:

```
Simulink Configuration Parameter settings for the model
'rtwdemo_pil_link_ts' and model
'rtwdemo_pil_component_mid1_link_ts' are incompatible.
The Link or Target product settings in the configuration set
for the two models result in different build folders
for the model reference code:

rtwdemo_pil_link_ts : slprj\ert_c167cs_sim\...
rtwdemo_pil_component_mid1_link_ts : slprj\ert_c167cs_hw\...

Please check that the two models have compatible Link and
Target settings.
```

To work around this problem, change to a clean work folder after changing your target preferences and before rebuilding. Alternatively, update all models and then they will rebuild correctly with the new settings.

Processor-in-the-Loop Issues

In this section...

"Generic PIL Issues" on page A-16
"On-Chip PIL Not Supported on ARM Hardware" on page A-16
"10-Second Pause on Termination of the CrossView Pro Debugger" on page A-16
"DSP563xx Link-Order Issue Can Cause PIL Application Failure" on page A-17
"No Support for TASKING Feature "Treat double as float"" on page A-17
"TASKING Optimization Settings May Cause Incorrect Cosimulation Results" on page A-18

Generic PIL Issues

See the Support Table section in the Real-Time Workshop Embedded Coder documentation for general PIL feature support information affecting the PIL block with Link products. See "SIL and PIL Simulation Support and Limitations".

On-Chip PIL Not Supported on ARM Hardware

For ARM processors the CrossView Pro instruction set simulator can be used for debugging and processor-in-the-loop (PIL) cosimulation, but there is currently no on-chip debugging or PIL support.

See "On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware" on page A-4 for solutions for this issue.

10-Second Pause on Termination of the CrossView Pro Debugger

When you terminate an instance of the CrossView Pro debugger application that was launched by Embedded IDE Link software, there is a pause of about 10 seconds before the CrossView Pro window closes. This 10-second pause is the intended behavior of CrossView Pro when acting as a COM server.

CrossView Pro pauses for the 10 seconds to wait for clients such as MATLAB to release their COM references.

DSP563xx Link-Order Issue Can Cause PIL Application Failure

When building PIL applications for DSP563xx you may see linker errors similar to the following example:

```
lk563 E208 (0): Found unresolved external(s):
            FDotProduct_s32s16            - (fuelsys0.a:fuelsys0.obj)
            FLook2D_S16_S16_S16_SAT       - (fuelsys0.a:fuelsys0.obj)
            FBINARYSEARCH_S16             - (fuelsys0.a:fuelsys0.obj)
            FINTERPOLATE_S16_S16_SAT      - (fuelsys0.a:fuelsys0.obj)
            FINTERPOLATE_EVEN_S16_S16_SAT - (fuelsys0.a:fuelsys0.obj)
wmk: *** action exited with value 1.
```

To resolve this issue, contact TASKING for a patch to make it possible to use the multipass option to rescan multiple libraries.

No Support for TASKING Feature "Treat double as float"

You can enable the feature in a TASKING project to treat the double-precision floating point datatype "double" as the single-precision floating point datatype "float". Usually, this means that double-precision floating point datatypes are represented in 4 bytes rather than 8 bytes.

PIL always assumes that the "double" datatype is represented normally. If you enable the "Treat double as float" override, PIL does not correctly transfer "double" datatypes between host and target, and unexpected data transfer errors occur during cosimulation. The default templates that ship with Embedded IDE Link software do not enable the override "Treat double as float" project option.

To resolve this issue:

- Do not use the option to treat "double" as "float". In this case, double precision floating point values are represented normally.
- Use the "single" datatype in Simulink rather than "double". In this case, the option to treat "double" as "float" will have no effect on PIL, because no "double" datatypes are used.

This is an example of the wider issue of problems caused by mismatching datatypes on host and target. For more details, see "Data Type Size Mismatch Issues (Embedded IDE Link)" in the Real-Time Workshop Embedded Coder documentation.

TASKING Optimization Settings May Cause Incorrect Cosimulation Results

Sometimes, you may observe differences between simulation and PIL cosimulation results. The code compiled and running in the TASKING environment may not always behave correctly, even when the generated code is correct. One cause of this issue, particularly with the TriCore toolset, is the compiler optimization configuration used to build the generated code.

If you see differences between simulation and PIL cosimulation results, to resolve this issue try setting the compiler optimization settings in the template projects to either `No optimization`, `Debug purpose`, or a similar equivalent for your TASKING toolset. Then, build the PIL algorithm and PIL application again and try repeating the cosimulation.

To create new template projects and modify their project settings see "Tutorial: Creating New Template Projects" on page 5-4.

Issues Using Real-Time Workshop Software Without Real-Time Workshop Embedded Coder Software

In this section...

"Real-Time Workshop grt.tlc-Based Targets Not Supported for PIL" on page A-19

""Save data to workspace" Causes Error" on page A-19

"DSP563xx Toolset Support Limitations" on page A-19

"Use ERT Target for Memory-Constrained Targets" on page A-20

"8051 GRT Limitations" on page A-20

Real-Time Workshop grt.tlc-Based Targets Not Supported for PIL

Real-Time Workshop "grt.tlc"-based targets are not supported for PIL.

To resolve this issue, use a Real-Time Workshop "ert.tlc"-based target.

"Save data to workspace" Causes Error

Simulink scope blocks with the "Save data to workspace" option checked cause a link error when building with GRT. This error occurs because this setting causes GRT to log data to a MAT-file during execution. However, MAT-file logging is not supported by Embedded IDE Link software.

When using ERT, Embedded IDE Link software makes sure the **MAT-file logging** configuration set option under **Real-Time Workshop** > **Interface** is not checked, and therefore this error is avoided.

DSP563xx Toolset Support Limitations

Only 16-bit mode for the DSP563xx Family is supported. Real-Time Workshop `grt.tlc`-based targets and the "GRT Compatible Call interface" option in the Real-Time Workshop Interface settings are not supported. This limitation is because of the nonstandard size of single- and double-precision floating-point datatypes on this architecture (`tmwtypes.h` does not compile).

You must use 16–bit mode.

Use ERT Target for Memory-Constrained Targets

Some targets such as the TASKING TriCore 1766B have memory constraints that can cause errors if you use the GRT target.

The 1766b has no external memory. You should use ERT rather than GRT when targeting this board, due to memory resource constraints. If you use the GRT target you may see compilation errors similar to the following example:

```
ltc E117: conflicting restriction for sections ".text.libc" and
".text.trapvec.000": absolute restrictions overlap
```

This problem occurs because the ERT (embedded real time) target is optimized for size and speed, while the GRT (generic real time) target is designed for ease of prototyping which incurs extra memory usage.

Use the ERT target for memory-constrained targets such as the TASKING TriCore 1766B.

See also “Linker Errors Due to Limited Memory” on page A-6.

8051 GRT Limitations

Working with the 8051 has some limitations when using GRT.

CrossView Pro Parameters

GRT application builds link against an example main (`grt_main.c`) file which includes a main function with `argc` and `argv` parameters for handling command-line arguments. When executing the application in CrossView Pro, these parameters are uninitialized and application execution terminates early. This behavior differs from that of other toolsets, where these parameters are initialized to 0 (`argc`) and the null pointer (`argv`).

To work around this issue on 8051, you can manually set `argc` to `0` in CrossView Pro before beginning execution.

Alternatively, you can create a library project for algorithm export that does not link against `grt_main.c` — see "Setting Build Action" on page 1-23 for more detail.

Signal Processing Blockset Software

The Signal Processing Blockset library fails to build for GRT models with the 8051 toolset. Certain datatypes required by the Signal Processing Blockset software, for example, real64_T, are not defined by Real-Time Workshop software for this configuration.

Use a Real-Time Workshop Embedded Coder, ERT-based target, rather than a GRT-based target.

Index

A

add build subdirectory suffix 1-18
Add Embedded IDE Link Configuration to
    Model 1-29

B

build action 1-17
    setting 1-23
build configuration 1-17
build process
    command line information 2-10
    folder structure 2-9
    overview 2-2
    shared libraries 2-6
    template projects 2-4
build subdirectory suffix 1-18

C

classes 2-14
components
    , project generator 2-13
    project generator 2-2
configuration options 1-16
configuration parameters
    pane 6-3
        Add build directory suffix 6-7
        Build directory suffix: 6-8
        Configure model to build PIL algorithm
            object code 6-13
        CrossView Pro handle name: 6-11
        EDE handle name: 6-9
        Export CrossView Pro handle to
            MATLAB base workspace 6-11
        Export EDE handle to MATLAB base
            workspace 6-9
        TaskingBuildAction 6-4
        TaskingConfiguration 6-6
        TaskingPILBlockAction 6-14
configuration sets 1-16
Configure model to build PIL algorithm object
    code 1-18
Create a New Model (configured for use with
    TASKING) 1-28
Create New Template Projects 1-28
CrossView Pro handle name 1-18

D

Demos 1-29

E

EDE handle name 1-18
Embedded IDE Link™ for use with Altium®
    TASKING®
    build process 2-1
    objects 2-13
Embedded IDE Link™ product
    PIL cosimulation 3-2
Embedded IDE Link™ software
    Embedded IDE Link Utilities for Use with
        TASKING dialog 1-27
    introduction 1-2
    limitations and tips A-1
    supported toolsets 1-7
    target preferences 1-10
    Tools menu 1-29
    tutorials 5-1
    user guide 1-9
Export CrossView Pro handle to MATLAB base
    workspace 1-18
Export EDE handle to MATLAB base
    workspace 1-18
Export handles 1-18

L

Launch and Test Communication with TASKING
    EDE 1-27

M

methods
  tasking.edeapi 2-22
  tasking.edeproject 2-24
  tasking.edeprojectspace 2-24
  tasking.xviewapi 2-24

N

new configuration
  creating 5-7

O

objects
  accessing properties 2-21
  calling methods 2-21
  creating 2-19
  demo 2-22
  finding methods 2-20
  finding properties 2-21
  list of methods 2-22
  obtaining method help 2-20
  terms 2-13
  using 2-15
Open Existing Template Projects 1-28
option sets 1-30
  tutorial 5-2
Options 1-29

P

PIL block
  creating 3-5
PIL block action 1-19
PIL cosimulation
  building and downloading 3-8
  coverage and profiling reports 3-13
  debugging 3-10
  overview 3-2
  profiling reports 3-15
Processor-in-the-Loop (PIL) Verification 1-18
project-based build process 2-4

R

Remove Embedded IDE Link Configuration from
  Model 1-29

S

Select Preconfigured Target Preference
  Settings 1-27

T

target preferences
  fields 1-13
  setting 1-10
Target Preferences 1-27
  Tools menu 1-29
Target Preferences Configuration 1-17
TASKING® CrossView Pro (Debugger)
  MATLAB® API 1-3
TASKING® EDE
  MATLAB® API 1-3
template projects 2-4
  creating 5-4
tutorial
  configuring existing models 5-9
  new configuration 5-7
  new template projects 5-4
  option sets 5-2

V

View, Modify, and Copy Configuration Sets via
  Model Explorer 1-28