

Link for TASKING[®]

For Use with Real-Time Workshop[®]

■ Modeling

■ Simulation

■ Implementation

User's Guide

Version 1



How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Link for TASKING User's Guide

© COPYRIGHT 2006 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

TASKING is a registered trademark of Altium Limited.

Other product or brand names are 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
September 2006 Online only

New for Version 1.0 (Release 2006a+)
Version 1.0.1 (Release 2006b)

Getting Started

1

What Is Link for TASKING?	1-2
Supported TASKING Toolsets	1-4
Support for Other Versions	1-4
Using This Guide	1-6
Setting Target Preferences	1-7
Target Preference Fields	1-9
Working with Configuration Sets	1-13
Setting Build Action	1-16
Link for TASKING Menus	1-18
Start Menu Items	1-18
Tools Menu Items	1-20
Option Sets	1-22
Link for TASKING Configuration Options	1-24
Known Limitations and Tips	1-28
Build Process	1-28
Processor-in-the-Loop (PIL)	1-34

Build Process

2

Build Process Overview	2-2
-------------------------------------	------------

Code Generation Process	2-2
Build Process	2-3
Memory Placement Example	2-3
Project-Based Build Process	2-4
Target Project Space	2-4
Template Projects	2-5
Relocation of Template Projects	2-5
How the Build Process Modifies the Relocated Template Project	2-6
Shared Libraries	2-7
Utility Function Generation: Shared Location	2-7
Supporting Multiple Shared Utility Function Locations: Build Subdirectory Name	2-8
Build Process — Directory Structure	2-10
Command Line Project Information	2-11

Objects

3

Objects for Link for TASKING	3-2
Classes	3-3
Using Objects	3-4
Creating an Object	3-4
Determining the Available Methods for a Class	3-6
Obtaining Help for a Class Method	3-6
Calling a Method	3-7
Determining the Available Properties for a Class	3-7
Accessing a Property	3-7
Objects Demo Example	3-7
List of Methods	3-9

Methods for Class tasking.edeapi	3-9
Methods for Class tasking.edeprojectspace	3-10
Methods for Class tasking.edeproject	3-10
Methods for Class tasking.xviewapi	3-10

Processor-in-the-Loop (PIL) Cosimulation

4

Overview of PIL Cosimulation	4-2
Why Use Cosimulation?	4-2
Definitions	4-3
How Cosimulation Works	4-4
 Creating a PIL Block	 4-5
 The PIL Cosimulation Block	 4-7
 Building, Running, and Debugging PIL Applications ..	 4-10
Building and Downloading PIL Applications	4-10
PIL Debugging	4-11
Coverage and Profiling Reports	4-13

Tutorials

5

Tutorial: Using Option Sets	5-2
 Tutorial: Creating New Template Projects	 5-4
Tutorial: Creating a New Configuration	5-6
 Tutorial: Configuring an Existing Model for Link for TASKING	 5-8
 Tutorial: Build Actions	 5-10

Getting Started

What Is Link for TASKING? (p. 1-2)	Introduces Link for TASKING® and its capabilities.
Supported TASKING Toolsets (p. 1-4)	TASKING toolsets supported by Link for TASKING.
Using This Guide (p. 1-6)	Suggested path through this document to get you up and running quickly with Link for TASKING.
Setting Target Preferences (p. 1-7)	Configuring Link for TASKING for use with specific development tools.
Working with Configuration Sets (p. 1-13)	A step-by-step example of configuring a Link for TASKING model for building with different toolchains.
Link for TASKING Menus (p. 1-18)	A quick guide to the functionality available in the Start and Tools menus, with links to instructions for tasks.
Option Sets (p. 1-22)	How to use preconfigured option sets to switch target settings.
Link for TASKING Configuration Options (p. 1-24)	A quick guide to Link for TASKING options in the Model Explorer with links to information on how to use these settings.
Known Limitations and Tips (p. 1-28)	A description of known limitations of Link for TASKING, with suggestions for workarounds.

What Is Link for TASKING?

Link for TASKING lets you build, test, and verify automatically generated code using MATLAB®, Simulink®, Real-Time Workshop®, and the TASKING integrated development environment. Link for TASKING 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 Real-Time Workshop Embedded Coder. A wide range of DSPs and 8-, 16- and 32-bit microprocessors and microcontrollers are supported including devices from Infineon, Renesas, and Freescale. Link for TASKING provides customizable templates for configuring hardware variants, automating MISRA C code checking, and controlling the build process.

With Link for TASKING, you can use MATLAB and Simulink to interactively analyze, profile and debug target-specific code execution behavior within TASKING. In this way, Link for TASKING 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.

Features include:

- Automated project-based build process
Automatically create and build projects for code generated by Real-Time Workshop or Real-Time Workshop Embedded Coder.
- Automated download and debugging
Rapidly and effortlessly debug generated code in the CrossView Pro debugger, using either the instruction set simulator or real hardware.
- Processor-in-the-loop (PIL) cosimulation
Use cosimulation techniques to verify generated code running in an instruction set simulator or real target environment.

- Highly customized code generation

Use Link for TASKING with any Real-Time Workshop System Target File (STF) to generate target-specific and optimized code.

- Highly customized 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.

- 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 CrossView Pro, 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 CrossView Pro.
- Set breakpoints, step through code, set parameters and retrieve profiling reports

Supported TASKING Toolsets

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

- Infineon TriCore: TASKING C/C++, CrossView Pro SIM for TriCore v2.3 r1 patch 1
- Infineon C166: TASKING Tools for C166/ST10 v8.6 r1
- Renesas M16C: TASKING Tools for M16C v3.1 r1 patch 2
- ARM: TASKING C Compiler for ARM v1.1 r1
- Freescale DSP563xx: TASKING Tools for DSP563xx v3.5 r3 patch 2
- 8051: TASKING Tools for 8051 v7.1 r3

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 Link for TASKING 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, you must first specify the location of your TASKING toolset in the Target Preferences (see “Setting Target Preferences” on page 1-7) then 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 Link for TASKING Product Support page for the latest information about toolchains qualified with the Link for TASKING. You may be able to obtain patches in order to use other toolsets.

Using This Guide

To get started with Link for TASKING:

- 1 Follow the instructions in “Setting Target Preferences” on page 1-7.
- 2 After you set target preferences, follow the instructions in “Working with Configuration Sets” on page 1-13 to see how to set up configurations using an example model.
- 3 Try the demos to gain experience using Link for TASKING. Access the demos in one of these ways:
 - Click the link: “Link for TASKING Demos.”
 - Select **Start > Simulink > Link for TASKING > Demos**.
 - Enter `demo('simulink', 'link for tasking')` at the MATLAB command line.
- 4 See “Link for TASKING Menus” on page 1-18 for a quick guide to the functionality available in the menus, with links to more information.

See the following chapters to learn about Link for TASKING features:

- Chapter 2, “Build Process” explains the Link for TASKING build process.
- Chapter 3, “Objects” explains how to create and use Link for TASKING objects.
- Chapter 4, “Processor-in-the-Loop (PIL) Cosimulation” describes how to use PIL cosimulation.
- Chapter 5, “Tutorials” contains instructions to show you how to create new configurations and template projects, how to use Link for TASKING with existing models, and how to use different build actions.

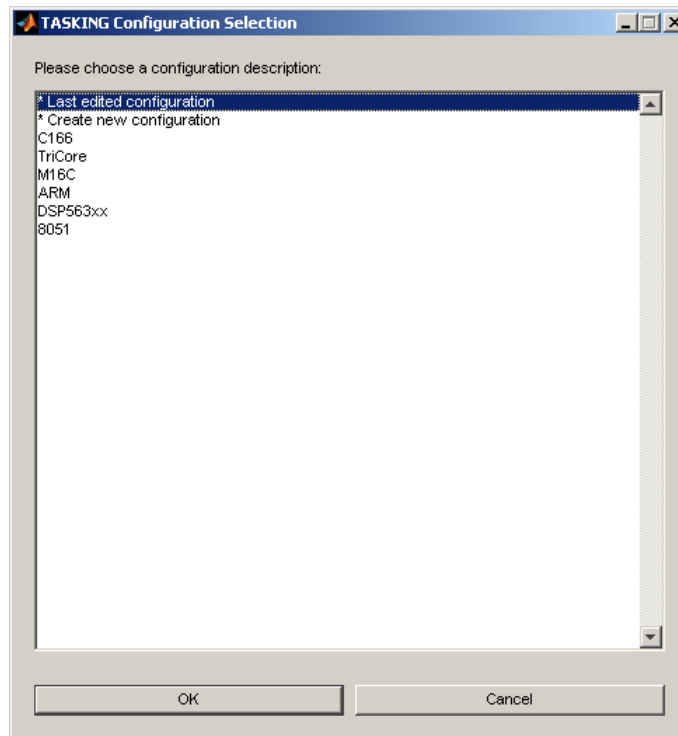
Setting Target Preferences

You must configure your target preferences to use Link for TASKING.

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

- 1 Select **Start > Simulink > Link for TASKING > TASKING Target Preferences**, or enter `tasking_edit_prefs`.

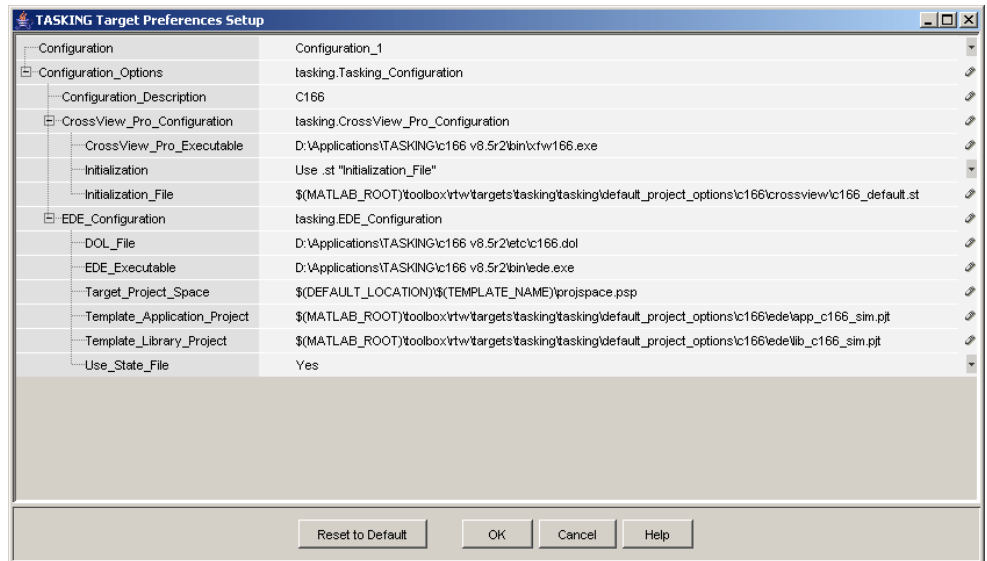
The TASKING Configuration Selection dialog box appears.



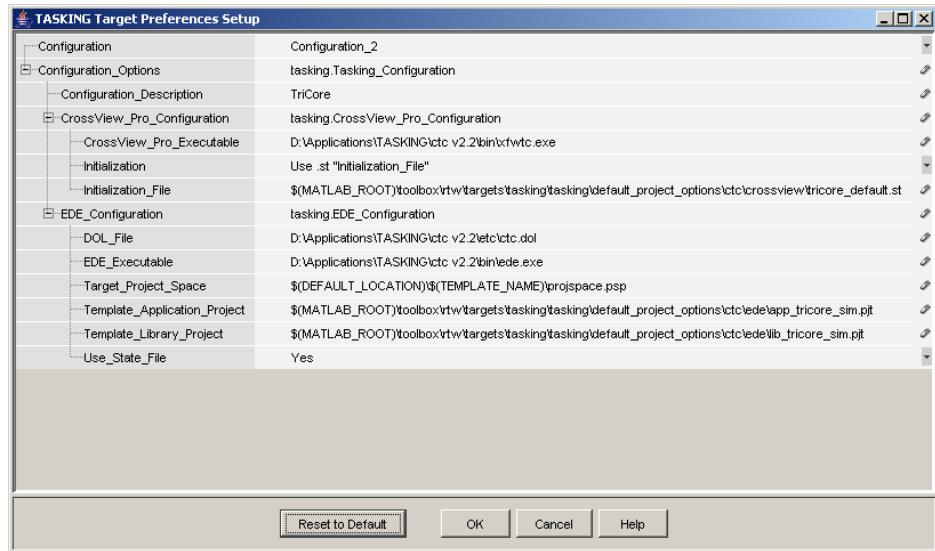
- 2 Select a predefined configuration from the list that matches your target, or select Create new configuration to create a new configuration, and click **OK**. For new configurations, see the tutorial section “Tutorial: Creating a New Configuration” on page 5-6.

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

- 3 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.



- 4 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 1-9, 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 1-13.

5 Click **OK** to dismiss the TASKING Target Preferences Setup dialog box.

The next section explains each target preference field.

Target Preference Fields

Open the Target Preference Setup dialog box by selecting **Start > Simulink > Link for TASKING > TASKING Target Preferences**, or enter `tasking_edit_prefs`.

- 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 “Tutorial: Creating a New Configuration” on page 5-6.

- Configuration_Description

The title of the configuration. After it is created, this title is the name that appears in the TASKING Configuration Description 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 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 Link for TASKING 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 1-22 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 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 `$(DEFAULT_LOCATION)\$(TEMPLATE_NAME)\projSPACE.psp`. The code generation process expands the `$(DEFAULT_LOCATION)` token with the build directory of the model, and the `$(TEMPLATE_NAME)` token with the name of the template application project. It is advisable to avoid changing this default setting.

- `Template_Application_Project`

When you build a Simulink model with Link for TASKING, 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 “Link for TASKING Menus” on page 1-18 for information on creating or opening template projects, and see “Template Projects” on page 2-5.

- `Template_Library_Project`

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

- `Use_State_File`

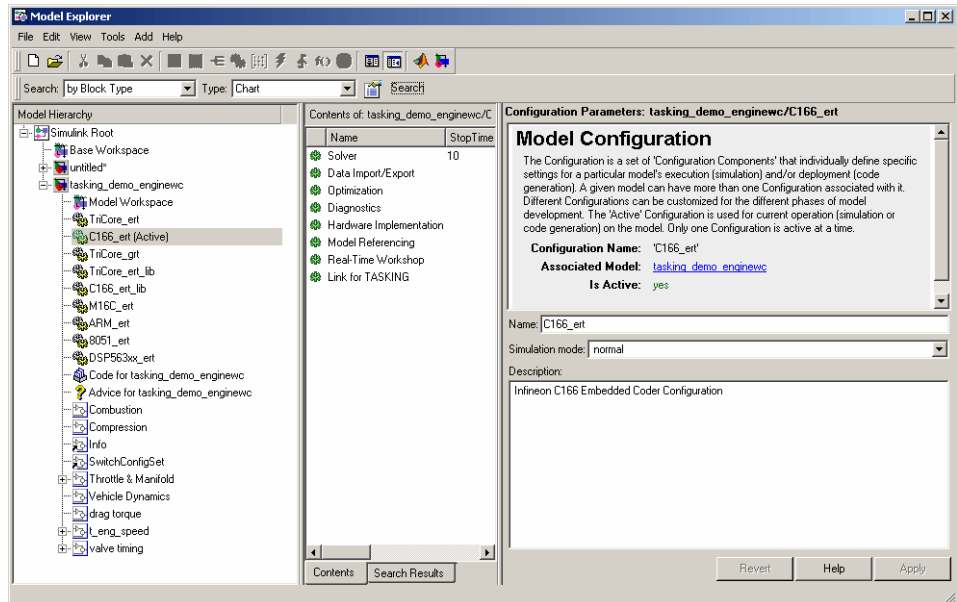
Opens the TASKING EDE in its last saved state. For more information, refer to your TASKING EDE documentation.

Working with Configuration Sets

Follow the steps in this example to see where to find and change Link for TASKING 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. See “Configuration Sets” in the Simulink documentation for more information.

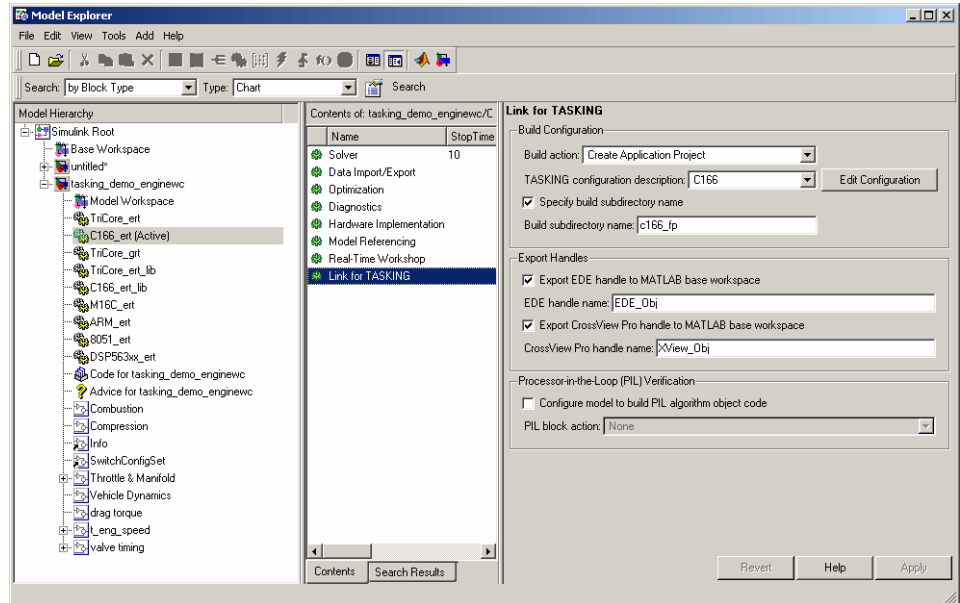
This example describes how to use Link for TASKING to build a project from a demo model using two different toolchains. The instructions refer to C166 and TriCore TASKING toolchains; adapt the instructions to your toolchain as appropriate.

- 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. The currently selected set is labeled (Active). Inspect the active configuration set.

- a The default active configuration set for this model is C166_ert. If you want to use a different target, right-click the configuration set that matches your target, and select **Activate**.
- b Click **Link for TASKING** to see the configuration settings, as shown following.



- c The **TASKING configuration description** 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).
 - i Click **Edit Configuration** to inspect your current target preferences.
 - ii 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 1-7.
 - iii Click **OK** to dismiss the TASKING Target Preferences Setup dialog box.

In the Link for TASKING demos, when you activate a configuration (e.g., C166_ert), the appropriate **Tasking configuration description** is automatically selected (e.g., C166). 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 “Tutorial: Creating a New Configuration” on page 5-6.

See “Link for TASKING Configuration Options” on page 1-24 for information on other Link for TASKING settings in the Configuration Parameters.

- d** Click **Real-Time Workshop** to see the selected system target file `ert.tlc`.

Note You can use a configuration set specifying any system target file with Link for TASKING.

- e** 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)
 - f** Close the Model Explorer.
- 3** 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.

If you have multiple toolchains, you have to set up your target preferences only once, then 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_ert` 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 demo models. See the “Link for TASKING Demos.”

To switch between simulator and hardware implementations for the same target configuration, you can use option sets. See “Option Sets” on page 1-22.

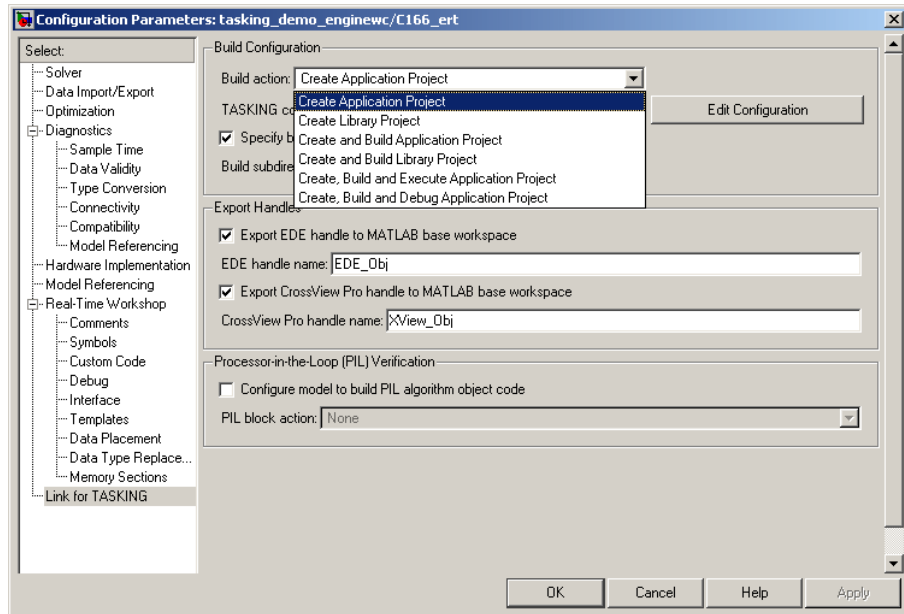
The next section describes using the build action setting in this example.

Setting Build Action

In this example, 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 **Link for TASKING** 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, CrossView Pro 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.

For more information on other build actions, see “Tutorial: Build Actions” on page 5-10.

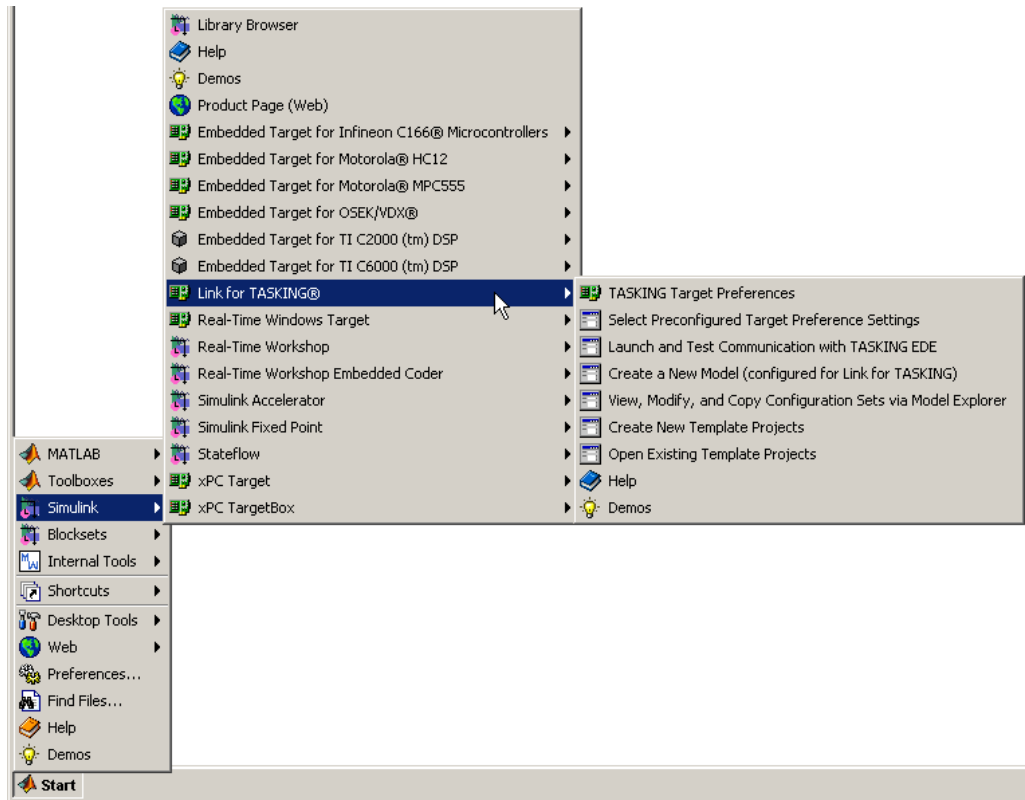
Link for TASKING Menu

This section describes the menu items, with links to instructions.

- “Start Menu Items” on page 1-18
- “Tools Menu Items” on page 1-20

Start Menu Items

Common tasks are available in the **Start** menu. Select **Start > Simulink > Link for TASKING**, as shown in the following figure, to see the following submenu options.



- **TASKING Target Preferences**

Opens the TASKING Configuration Selection dialog box, and after you choose a configuration to match your target (e.g., TriCore), you can edit the TASKING Target Preferences Setup 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 Link for TASKING. See “Setting Target Preferences” on page 1-7.

- **Select Preconfigured Target Preference Settings**

Opens the TASKING 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 1-22.

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

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

- **Create a New Model (configured for Link for TASKING)**

Creates a new untitled Simulink model, with Link for TASKING configuration set options already added. You can also configure an existing model by selecting the Simulink model menu item **Tools > Link for TASKING > Add Link for TASKING 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 Link for TASKING product ships with preconfigured application and library template projects for the default configurations in the TASKING Preferences. 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 2-5 for more information.

This option opens the TASKING Configuration Selection dialog box. Choose a configuration and click **OK**, and Link for TASKING launches the appropriate TASKING EDE and creates new template projects for a specific TASKING configuration. You are prompted to choose a project directory, a template name, and an option set. See “Option Sets” on page 1-22 for more details. `app_template_name.pjt` and `lib_template_name.pjt` are created for the configuration you selected.

- **Open Existing Template Projects**

Opens the 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)`.

- **Demos**

Opens the Link for TASKING Demos page in the Help browser.

Tools Menu Items

In a Simulink model, you can access Link for TASKING items in the **Tools** menu. Select **Tools > Link for TASKING** to see the following submenu items.

- **TASKING Target Preferences**

As in the **Start** menu, opens the TASKING Configuration Selection dialog box, and once you have chosen a configuration, you can edit the TASKING Target Preferences Setup dialog box. You must set up your target preferences before you can use Link for TASKING. See “Setting Target Preferences” on page 1-7.

- **Add Link for TASKING Configuration to Model**

Adds Link for TASKING 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 Link for TASKING Configuration from Model**

Removes Link for TASKING configuration options from the model's configuration parameters.

- **Options**

Opens the Configuration Parameters dialog box to show Link for TASKING options. See “Link for TASKING Configuration Options” on page 1-24.

Option Sets

Option sets are preconfigured settings to specify the target configuration for the 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 either

- Use option sets to switch between default target configurations, or
- Use option sets 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 Link for TASKING Demos.
 - `tricore_1796b`: Infineon TriCore 1796b hardware configuration.
 - `tricore_1766b`: Infineon TriCore 1766b hardware configuration.
- Infineon C166:
 - * `c167cs_sim`: Infineon C167CS instruction set simulator configuration.
 - `c166_sim`: Default instruction set simulator configuration.
 - `c167cs_hw`: As `c167cs_sim`, but targeting hardware rather than simulator.

- Renesas M16C
 - * m16c_sim: Default instruction set simulator configuration.
 - r8ctiny_sim: Renesas R8C Tiny instruction set simulator configuration.
- Freescale DSP563xx:
 - * dsp563xx_sim: DSP563xx Family, 16-bit memory model, instruction set simulator configuration.
 - dsp566xx_sim: DSP566xx Family instruction set simulator configuration.
- ARM:
 - * arm_sim: Default instruction set simulator configuration.
 - arm_sim_big_endian: As arm_sim, but in big-endian mode.
- 8051:
 - * i8051_sim: Default, large memory model, no language extensions, floating point, instruction set simulator configuration.

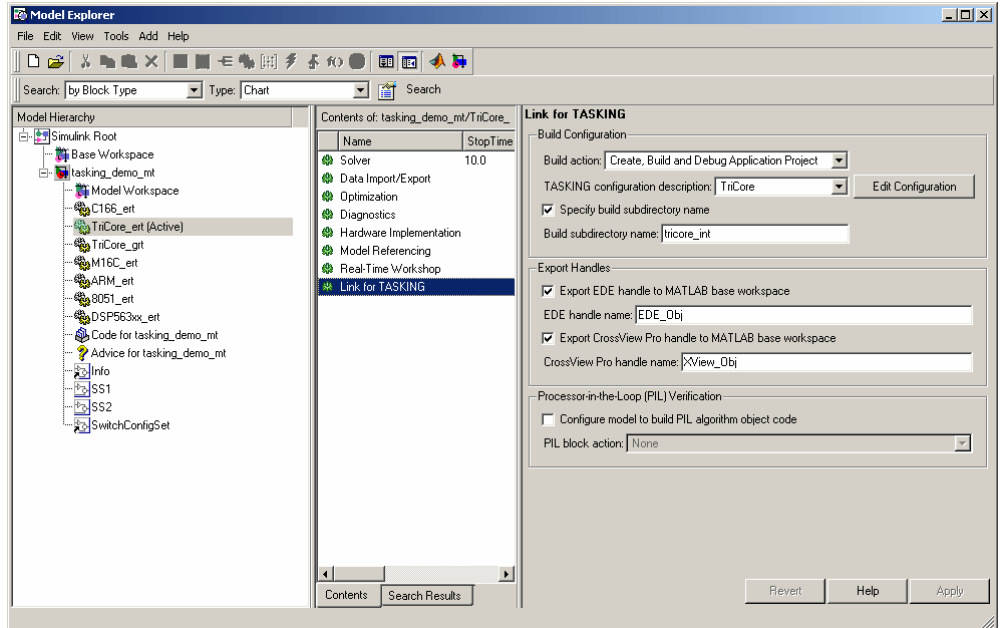
Link for TASKING Configuration Options

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

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

- **Simulation > Configuration Parameters** in a model
- **Tools > Link for TASKING > Options** in a model
- **View > Model Explorer** in a model
- **Start > Simulink > Link for TASKING > View, Modify and Copy Configuration Sets via Model Explorer** in MATLAB

Click Link for TASKING 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 TASKING EDE and then stop, or you can also choose to build, execute, or debug. See “Tutorial: Build Actions” on page 5-10 for more details.

- **TASKING configuration description**

Select target preference configurations. The names correspond to the Configuration Description for each configuration in the TASKING Target Preferences Setup dialog box. Click **Edit Configuration** to open the TASKING Target Preferences Setup dialog box for the currently selected configuration. See “Working with Configuration Sets” on page 1-13.

- **Specify build subdirectory name**

Select the check box to specify a subdirectory name and avoid “shared utility function” code generation errors. Prior to code generation, Link for TASKING changes to the specified build subdirectory to avoid conflicts over the shared utility location. Clear this check box to use the default Real-Time Workshop build without using a subdirectory — not using a subdirectory may result in rebuilding shared libraries unnecessarily. See “Shared Libraries” on page 2-7 and particularly “Supporting Multiple Shared Utility Function Locations: Build Subdirectory Name” on page 2-8 for details.

- **Build subdirectory name**

Enter a name for the subdirectory in the edit box.

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 Chapter 3, “Objects” 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 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 Chapter 4, “Processor-in-the-Loop (PIL) Cosimulation” for more information on using PIL settings.

Known Limitations and Tips

The following issues are known limitations with Link for TASKING, with suggestions for workarounds where possible.

Build Process

- “EDE Is Slow, Unresponsive, or Crashes” on page 1-28
- “Signal Processing Blockset Library Build Failures” on page 1-29
- “Model Reference Is Not Supported” on page 1-30
- “Real-Time Workshop “grt.tlc”-Based Targets Are Only Supported for 32-Bit Targets” on page 1-30
- “Use ERT Target for TASKING TriCore 1766B” on page 1-31
- “Memory Block Freed Twice Error” on page 1-31
- “8051 EDE Cannot Compile Files with Long Names” on page 1-31
- “8051 Compiler Bug: Assertion Failure” on page 1-32
- “ARM GRT Build Failure” on page 1-32
- “DSP563xx Toolset Support Limitations” on page 1-32
- “ “Create, Build and Execute Application Project” Build Action Fails ” on page 1-33
- “C166 Toolset Warnings” on page 1-34

EDE Is Slow, Unresponsive, or Crashes

Tool Suites: All

Problem: 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.

Workaround: 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` directory (`$TASKINGRootDir\bin`)

Signal Processing Blockset Library Build Failures

The following problems have been found with Signal Processing Blockset (“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.
Workaround: Disable floating point support in the model.

- With 8051, when trying to build DSP libraries, you may see the following errors with floating- and fixed-point versions:

```
TASKING program builder v7.1r3 Build 076 SN 00123456
Compiling g711_enc_a_sat_rt.c
cc51 S533: D:\work_dirs\tasking_bugs\8051_fixed_point_dsplib\
g711_enc_a_sat_rt.c: line 34: assertion failed - please report
wmk: *** action exited with value 2.
wmk: "g711_enc_a_sat_rt.src" removed.
```

```
TASKING program builder v7.1r3 Build 076 SN 00123456
Compiling burg_a_c_rt.c
wmk: *** action exited with value -1073741571.
wmk: "burg_a_c_rt.src" removed.
```

This limitation is a known issue with the 8051 compiler. Workaround: none.

- With ARM, when trying to build DSP libraries, you may see the following errors with floating- and fixed-point versions:

```
TASKING program builder v1.1r1 Build 078 SN 00123456
Compiling "..\..\..\..\..\..\..\..\aetargets with spaces\matlab\
```

```
toolbox\rtw\dspblks\c\dspendian\is_little_endian_rt.c"
carm S917: internal consistency check failed - please report
wmk: *** action exited with value 1.
```

```
TASKING program builder v1.1r1 Build 078 SN 00123456
Compiling "..\..\..\..\..\..\aetargets with spaces\matlab\
toolbox\rtw\dspblks\c\dspendian\is_little_endian_rt.c"
carm S917: internal consistency check failed - please report
wmk: *** action exited with value 1.
```

This limitation is a known issue with the ARM compiler. Workaround: none.

Model Reference Is Not Supported

Tool Suites: All

Problem: Model reference is not yet supported by Link for TASKING. An informative error is provided.

Workaround: None.

Real-Time Workshop "grt.tlc"-Based Targets Are Only Supported for 32-Bit Targets

Tool Suites: Infineon C166, Renesas M16C, 8051, DSP563xx

Real-Time Workshop "grt.tlc"-based targets are not supported for non-32-bit targets. If you use an unsupported combination you see an error of this form:

```
Error using ==> RTW.makertw.PCGHook
Error encountered while executing PostCodeGenCommand:
Error using ==> tasking_post_code_gen_hook>i_processBuildArgs
This model requires support for non-finite floating point values
("rt_nonfinite.c").
"rt_nonfinite.c" only compiles on targets with at least a 32-bit
word size.
However, this target has a word size of only: 16 bits.
To avoid this error you can switch to an ERT-based target and
uncheck "non-finite numbers" in the RTW Interface configuration
settings, however you will not be able to use non-finite
```

floating point values in the model.

Workaround: Use a Real-Time Workshop “ert.tlc”-based target.

Use ERT Target for TASKING TriCore 1766B

The 1766b has no external memory. You should use ERT rather than GRT when targeting this board, due to memory resource constraints. 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.

If you use the GRT target you may see compilation errors like the following:

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

Memory Block Freed Twice Error

Occasionally, when the Link for TASKING 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. Consider moving the model to a shorter directory name, or renaming the model, subsystem, or both to use shorter names.

8051 Compiler Bug: Assertion Failure

When building 8051 projects you may see the following error:

```
TASKING program builder v7.1r3 Build 076 SN 00123456

Compiling compilerassertion.c

cc51 S518: D:\Applications\tasking\8051\v7.1r3\examples\banksw\
compilerassertion.c: line 23: assertion failed - please report

wmk: *** action exited with value 2.

wmk: "compilerassertion.src" removed.
```

This limitation is a known issue with the 8051 compiler. Workaround: None.

ARM GRT Build Failure

With ARM, when building with the grt system target file, you may see the following error:

```
TASKING program builder v1.1r1 Build 078 SN 00123456
Compiling "..\..\slprj\grt\sharedutils\rt_nonfinite.c"
carm S917: internal consistency check failed - please report
wmk: *** action exited with value 1.
```

This limitation is a known issue with the ARM compiler. Workaround: None.

DSP563xx Toolset Support Limitations

The following limitations affect use of the DSP563xx Toolset:

- Only 16-bit mode for the DSP563xx Family is supported. As for other 16-bit targets, Real-Time Workshop “grt.tlc”-based targets are not supported; for this toolset the "GRT Compatible Call interface" option in the Real-Time Workshop Interface settings is also 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.

- PIL is not supported for the DSP563xx Toolset because it is a word addressable architecture and this is not yet supported by PIL. Only byte addressable architectures are supported.
- 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.

“Create, Build and Execute Application Project” Build Action Fails

Tool Suites: Renesas M16C

Executing the application project, rather than debugging (via “Create, Build and Debug Application Project”) does not work correctly, because the CrossView Pro Simulator does not know the start address when debugging information is not loaded. The application does not execute.

Workaround: 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.

Processor-in-the-Loop (PIL)

The following issues affect the use of PIL:

- “10-Second Pause on Termination of the CrossView Pro Debugger” on page 1-35
- “TASKING TriCore 1766B PIL Limitation” on page 1-35
- “8051 Link-Order Issue Can Cause PIL Application Failure” on page 1-35
- “8051 PIL Timeout Errors” on page 1-36
- “Buses and Mux Signals Not Supported at PIL Component Boundary” on page 1-36
- “Signals with Custom Storage Classes Not Supported at PIL Component Boundary” on page 1-36
- “Continuous Sample Times Not Supported” on page 1-36
- “Real-Time Workshop “grt.tlc”-Based Targets Not Supported” on page 1-36
- “Enabled / Triggered Subsystems Are Not Supported” on page 1-36
- “No Support for TASKING Feature “Treat double as float”” on page 1-37
- “TASKING Optimization Settings May Cause Incorrect Cosimulation Results” on page 1-37
- “Export Functions Feature Is Not Supported” on page 1-38

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 Link for TASKING, 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.

TASKING TriCore 1766B PIL Limitation

The demos “tasking_demo_pil_toplevel_testharness” and “tasking_demo_pil_library_testharness” do not work correctly for PIL, because of a TASKING issue relating to setting breakpoints. The PIL application does not download correctly in CrossView Pro and causes a 60-second pause in MATLAB before the following error occurs:

```
Error using ==> tasking.xviewapi.executeAndWait
Command "s", sequence number 14, timed out after 60 seconds.
```

Other PIL demos such as tasking_demo_autotrans do work correctly. However, customers' PIL models on this platform may run into the same issue.

8051 Link-Order Issue Can Cause PIL Application Failure

When building PIL applications for 8051 you may see linker warnings similar to the following:

```
link51 W001: unresolved external symbol
  '_?BINARYSEARCH_S16', module t_fuelsys.obj
link51 W001: unresolved external symbol
  '_?DotProduct_s32s16', module t_fuelsys.obj
link51 W001: unresolved external symbol
  '_?INTERPOLATE_S16_S16_SAT', module t_fuelsys.obj
```

If such a message appears, the PIL block reports that an error has occurred during cosimulation.

Workaround: if you encounter this you can contact TASKING for a patch to make it possible to use the multipass option to rescan multiple libraries.

8051 PIL Timeout Errors

See limitation and workaround details in .

Buses and Mux Signals Not Supported at PIL Component Boundary

Buses and MUX Signals are not supported at the PIL component boundary.

Workaround: None.

Signals with Custom Storage Classes Not Supported at PIL Component Boundary

Signals with Custom Storage Classes are not supported at the PIL component boundary.

Workaround: None.

However, note that the standard noncustom storage classes, like ExportedGlobal, are supported.

Continuous Sample Times Not Supported

Continuous sample times are not supported by PIL. If you encounter this you see the following error:

```
??? Processor-in-the-Loop (PIL) does not support continuous  
time. Please uncheck "continuous time" in the RTW Interface  
configuration set options or disable PIL.
```

Workaround: None. You must use discrete sample times.

Real-Time Workshop “grt.tlc”-Based Targets Not Supported

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

Workaround: Use a Real-Time Workshop “ert.tlc”-based target.

Enabled / Triggered Subsystems Are Not Supported

Enabled / Triggered subsystems are not supported for PIL.

Workaround: None.

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 cosimulation errors occur. The default templates that ship with Link for TASKING do not enable the override.

Workarounds:

- 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.

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.

Workaround: If you see differences between simulation and PIL cosimulation results, 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.

Export Functions Feature Is Not Supported

The Real-Time Workshop feature “Export Functions” is not supported.

Workaround: None.

Build Process

Build Process Overview (p. 2-2)	Understanding the build process.
Project-Based Build Process (p. 2-4)	About projects and target project space.
Template Projects (p. 2-5)	About template projects.
Shared Libraries (p. 2-7)	About shared libraries and build subdirectory names.
Build Process — Directory Structure (p. 2-10)	Explains the build process directory structure and how to locate files.

Build Process Overview

Link for TASKING provides a customized build process that is designed to work with the highly customized code generation process provided by Real-Time Workshop.

To explain the separation of duties between Real-Time Workshop and Link for TASKING, 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 Link for TASKING 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 Link for TASKING 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.

Link for TASKING 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. Link for TASKING 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 Link for TASKING 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.

Link for TASKING 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

Link for TASKING 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 name of the template application project (see `$(TEMPLATE_NAME)` token) that triggered the build process, as well as the current directory at the time the build process is invoked (see `$(DEFAULT_LOCATION)` token).

Template Projects

Template projects are regular TASKING EDE projects that are used by Link for TASKING 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, Link for TASKING 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 Link for TASKING 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 is 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

Link for TASKING 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 directory. 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 directory

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 Subdirectory Name

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 directory as the location for generated files. In this location, it supports only a single shared utilities directory for each system target file. Therefore, it is possible for conflicts over the contents of the shared utility directory to occur.

For example, this 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.

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 problem, Link for TASKING provides the “Specify Build Subdirectory Name” and “Build Subdirectory Name” settings.

During a Link for TASKING build, if the **Specify Build Subdirectory Name** check box is enabled then the name specified in the **Build Subdirectory Name** setting is used as the name of a directory to change to from the current working directory location. If this directory does not exist, it is created automatically.

Therefore, specifying the same build subdirectory name for a similar set of models allows them to generate code into their own working directory, avoiding conflict with other models, while still allowing a shared utilities directory.

At the end of the build process, the original working directory is restored.

This feature of Link for TASKING removes the need for the user to manually manage changing directories to avoid shared utility directory conflicts.

See the demo models for examples of using this setting: Link for TASKING Demos.

Build Process – Directory Structure

The following table shows the typical directories that are created, relative to the current working directory, during the Real-Time Workshop code generation process and the Link for TASKING build process. Library files are specific to library builds.

Note If the Link for TASKING Build Subdirectory option is being used, then the directories in the table are relative to the build subdirectory.

Directory	Contents
<code>\$(TEMPLATE_NAME)\pjt_\$(CODEGEN)</code>	Main application project: <code>\$(CODEGEN).pjt</code> and associated files (only for application builds).
<code>\$(TEMPLATE_NAME)\pjt_exp_ - \$(CODEGEN)</code>	Main library project: <code>exp_\$(CODEGEN).pjt</code> and associated files (only for library builds).
<code>\$(TEMPLATE_NAME)\pjt_rtwlib</code> (if required)	Real-Time Workshop library project: <code>rtwlib.pjt</code> and associated files.
<code>\$(TEMPLATE_NAME)\pjt_rtwshared</code> (if required)	Shared utilities library project: <code>rtwshared.pjt</code> and associated files.
Key	
<code>\$(CODEGEN)</code>	Real-Time Workshop code generation directory name
<code>\$(TEMPLATE_NAME)</code>	Token expanded from the name of the template application project in the target preferences. If the project name is prefixed with “user_” this token is removed. <code>\$(CONFIG_DESC)</code> is a valid alternative, which expands to the name of the TASKING configuration description.

See the next section, “Command Line Project Information” on page 2-11, 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:

ADD_MDL_NAME_TO_GLOBALS=1
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
```


Objects

Objects for Link for TASKING
(p. 3-2)

Classes (p. 3-3)

Using Objects (p. 3-4)

List of Methods (p. 3-9)

Introduction and definitions.

Classes provided with Link for TASKING.

How to create objects and find methods and properties.

Tables showing the methods available for Link for TASKING objects.

Objects for Link for TASKING

Link for TASKING 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 Link for TASKING 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 Link for TASKING objects. See “Objects Demo Example” on page 3-7 for an example demonstrating some basic capabilities of Link for TASKING objects.

Classes

The following table shows the different classes that are provided with Link for TASKING.

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

Using Objects

The topics in this section are:

- 1** “Creating an Object” on page 3-4
- 2** “Determining the Available Methods for a Class” on page 3-6
- 3** “Obtaining Help for a Class Method” on page 3-6
- 4** “Calling a Method” on page 3-7
- 5** “Determining the Available Properties for a Class” on page 3-7
- 6** “Accessing a Property” on page 3-7
- 7** “Objects Demo Example” on page 3-7

Creating an Object

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

```
tasking_help <classname>.<constructorname>
```

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

```
tasking_help tasking.edeapi.edeapi
```

For the `tasking.edeprojectspace` class, enter

```
tasking_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” on page 3-4), 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 3-9.

Obtaining Help for a Class Method

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

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

```
tasking_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 3-9.

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 [Link for TASKING Demos](#).

This example provides step-by-step instructions for using [Link for TASKING](#) 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 3-9
- “Methods for Class `tasking.edeprojectspace`” on page 3-10
- “Methods for Class `tasking.edeproject`” on page 3-10
- “Methods for Class `tasking.xviewapi`” on page 3-10

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`

<code>closeIDE</code>	<code>getOptionSetNames</code>
<code>disp</code>	<code>getProject</code>
<code>display</code>	<code>getProjectSpace</code>
<code>edeapi</code>	<code>getTargetProject</code>
<code>exec</code>	<code>getToolchainInfo</code>
<code>execApiMacro</code>	<code>newProject</code>
<code>execRetNumeric</code>	<code>newProjectSpace</code>
<code>execRetString</code>	<code>newProjectTemplates</code>
<code>getCreatedEDEProcess</code>	<code>newProjectTemplatesViaUI</code>
<code>getOptionSet</code>	<code>newTempProjectSpaceIfNoneOpen</code>
<code>open</code>	<code>processTemplateProject</code>
<code>openProjectTemplates</code>	<code>validateToolchainDirectory</code>
<code>pwd</code>	

Methods for Class `tasking.edeprojectspace`

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

Methods for Class `tasking.edeproject`

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

Methods for Class `tasking.xviewapi`

<code>addBreakpointCallback</code>	<code>getEventReporting</code>
<code>getFunctionConfiguration</code>	<code>debug</code>
<code>disp</code>	<code>halt</code>

<code>removeBreakpointCallbacks</code>	<code>display</code>
<code>isRunning</code>	<code>setEventReporting</code>
<code>downloadAndRun</code>	<code>execute</code>
<code>xviewapi</code>	<code>executeAndWait</code>
<code>getCommandResponse</code>	

Processor-in-the-Loop (PIL) Cosimulation

Overview of PIL Cosimulation
(p. 4-2)

Creating a PIL Block (p. 4-5)

The PIL Cosimulation Block (p. 4-7)

Building, Running, and Debugging
PIL Applications (p. 4-10)

Defining processor-in-the-loop (PIL)
Cosimulation.

How to create a PIL block.

Describing the Simulink block
interface to PIL.

How to use the PIL block to build,
download, cosimulate, and debug
PIL applications.

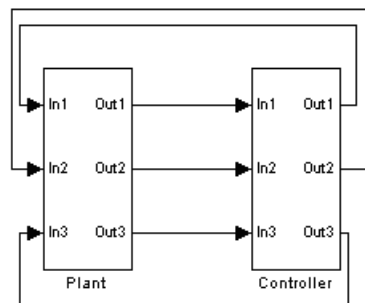
Overview of PIL Cosimulation

Link for TASKING supports processor-in-the-loop (PIL) cosimulation, a technique that is 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.

During the Real-Time Workshop Embedded Coder code generation process, Link for TASKING can create a PIL block from one of several Simulink components including a model, a subsystem in a model, or subsystem in a library. You then place the generated PIL block inside a Simulink model that serves as the test harness and run tests to evaluate the target-specific code execution behavior.

Why Use Cosimulation?

PIL cosimulation is particularly useful for simulating, testing, and validating a controller algorithm in a system comprising a plant and a controller. In classic closed-loop simulation, Simulink and Stateflow® model such a system as two subsystems and the signals transmitted between them, as shown in this block diagram.



Your starting point in developing a plant/controller system is to model the system as two subsystems in closed-loop simulation. As your design progresses, you can use Simulink external mode with standard Real-Time

Workshop targets (such as GRT or ERT) to help you model the control system separately from the plant.

However, these simulation techniques do not help you to account for restrictions and requirements imposed by the hardware (e.g., limited memory resources, or behavior of target-specific optimized code). When you finally reach the stage of deploying controller code on the target hardware, you may need to make extensive adjustments to the controller system. After these adjustments are made, your deployed system may diverge significantly from the original model. Such discrepancies can create difficulties if you need to return to the original model and change it.

PIL cosimulation addresses these issues by providing an intermediate stage between simulation and deployment. The term *cosimulation* reflects a division of labor in which Simulink models the plant, while code generated from the controller subsystem runs on the actual target hardware. In a PIL cosimulation, the target processor participates fully in the simulation loop — hence the term *processor-in-the-loop*.

Definitions

PIL Algorithm

The algorithmic code (e.g., the control algorithm) to be tested during the PIL cosimulation. The PIL algorithm resides in compiled object form to allow verification at the object level.

PIL Application

The executable application to be run on the target platform. The PIL application is created by linking the PIL algorithm object code with some wrapper code (or test harness) that provides an execution framework that interfaces to the PIL algorithm.

The wrapper code includes the `string.h` header file so that the `memcpy` function is available to the PIL application. The PIL application uses `memcpy` to facilitate data exchange between Simulink and the cosimulation target.

Note Whether the PIL algorithm code under test uses `string.h` is independent of the use of `string.h` by the wrapper code, and is entirely dependent on the implementation of the algorithm in the generated code.

How Cosimulation Works

In a PIL cosimulation, Real-Time Workshop generates efficient code for the PIL algorithm. This code runs (in simulated time) on a target platform. The plant model remains in Simulink without the use of code generation.

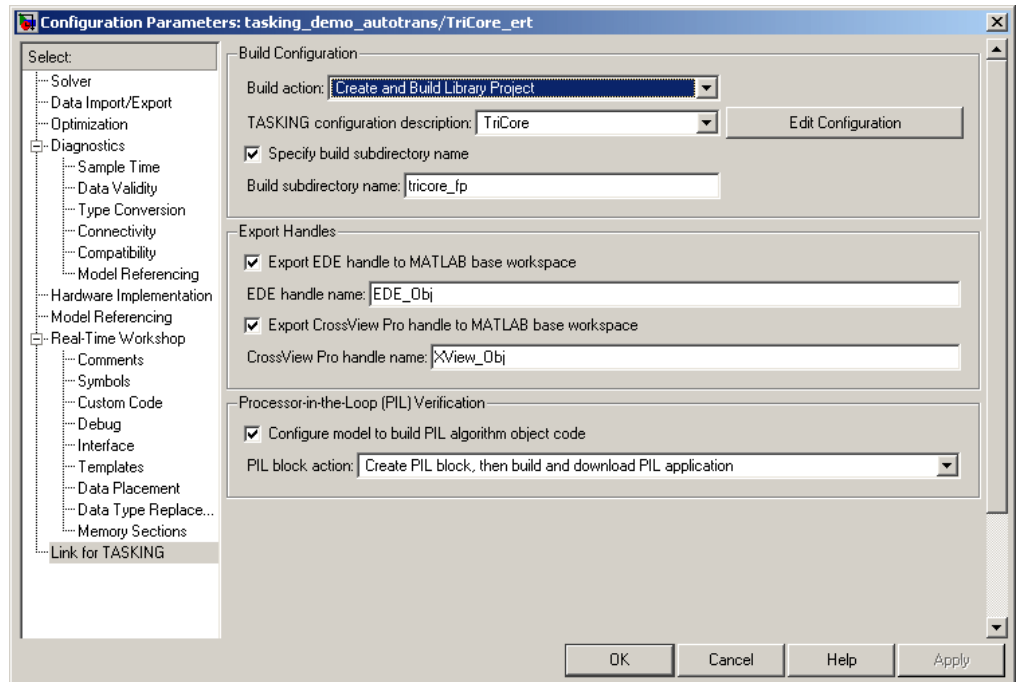
During PIL cosimulation, Simulink simulates the plant model for one sample interval and exports the output signals (Yout of the plant) to the target platform via the CrossView Pro debugger. When the target platform receives signals from the plant model, it executes the PIL algorithm for one sample step. The PIL algorithm returns its output signals (Yout of the algorithm) computed during this step to Simulink, via the CrossView Pro debugger. At this point, one sample cycle of the simulation is complete and the plant model proceeds to the next sample interval. The process repeats and the simulation progresses.

PIL tests do not run in real time. After each sample period, the tests halts to ensure that all data has been exchanged between the Simulink test harness and object code. You can then check functional differences between the model and generated code. During a PIL test, you can use the TASKING debugger to set breakpoints, step through the code, and watch variables.

After the test, Link for TASKING returns execution profiling and code coverage reports to MATLAB for your review. See “Coverage and Profiling Reports” on page 4-13 for more information.

Creating a PIL Block

The PIL settings can be found in the Configuration Parameters dialog box under the Link for TASKING 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.

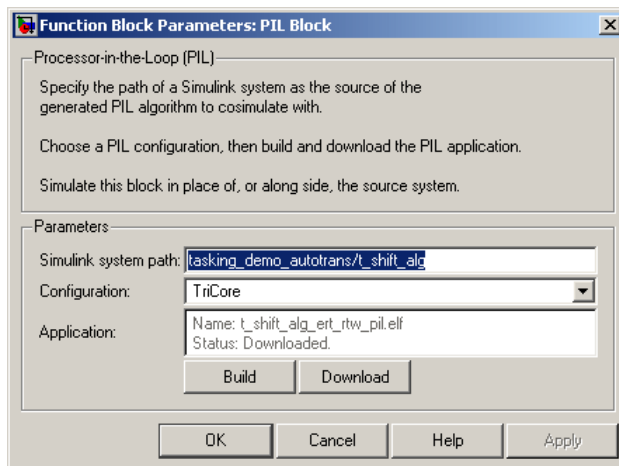
See “Building, Running, and Debugging PIL Applications” on page 4-10 for more details.

The PIL Cosimulation Block

The PIL cosimulation block is the Simulink block interface to PIL. The Simulink inputs and outputs of the PIL cosimulation block are configured to match the input and output specification of the PIL algorithm.

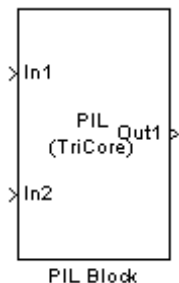
The block is a basic building block that allows you to:

- Select a PIL algorithm
- Choose a PIL configuration
- Build and download a PIL application
- Run a PIL cosimulation



See the next section, “Building, Running, and Debugging PIL Applications” on page 4-10 for instructions for using the PIL block.

The PIL block takes the same shape and signal names as the parent subsystem, like those in the following example. This inheritance is convenient for copying the PIL block into the model to replace the original subsystem for cosimulation.



Link for TASKING creates PIL blocks with both the "Simulink system path" and "Configuration" properties automatically configured. These parameters have the following characteristics:

Simulink system path — Allows you to select a PIL algorithm. You specify the path of a Simulink system (model or subsystem) as the source of the generated PIL algorithm to use for cosimulation.

The Simulink system path is the full path to the system and "/" must be escaped to "\". For example, a subsystem named "fuel/sys" inside a model named "tasking_demo_fuelsys" would have the escaped system path:

```
tasking_demo_fuelsys/fuel//sys
```

The correct system path can be obtained by clicking on the system and then running the gcb command. In this example,

```
>> gcb
ans =
tasking_demo_fuelsys/fuel//sys
```

Configuration — Allows you to specify a PIL configuration to use for building the PIL application and running the subsequent cosimulation. 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 Link for TASKING requires this information to communicate with the PIL application.
- 2** The configuration must be compatible with the TASKING configuration description 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.

Building, Running, and Debugging PIL Applications

This section includes the following topics:

- “Building and Downloading PIL Applications” on page 4-10
- “PIL Debugging” on page 4-11
- “Coverage and Profiling Reports” on page 4-13

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.

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 Link for TASKING demo models for examples with instructions to enable you to build and download PIL blocks and use them in cosimulation.

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 (usually set on the `pilaction` 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 Link for TASKING, 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.

Coverage and Profiling Reports

After you download the PIL application and run a cosimulation, you can view reports in MATLAB. The reports available depend on the target configuration. For example, for C166 Simulator you can view C code coverage, profiling and cumulative profiling reports. Messages at the command line detail which reports are available with hyperlinks, 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)
```

Click the variable name hyperlinks (e.g., [pil_coverage_report](#)) to view the reports, similar to the two following examples:

```
pil\_coverage\_report =

Module:    ..\..\fuelsys0_ert_rtw_pil\pil_interface.c    81%
Function:  pilInitialize                               77%
Function:  initUDataProcessing                        76%
Function:  processUData                               100%
Function:  checkDataProcessingComplete                100%
Function:  pilStep                                   71%
Function:  initYDataProcessing                        76%
Function:  processYData                               100%
Function:  pilTerminate                               75%
Module:    ..\..\..\..\..\..\aetargets with spaces\matlab\
toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_lib.c    90%
Function:  getNextSymbol                             100%
Function:  processData                               90%
Function:  resetLibSymbolState                       100%
Function:  checkDataProcessing                       78%
```

```

Module:  ..\..\..\..\..\aetargets with spaces\matlab\
toolbox\rtw\targets\tasking\tasking\tasking_pil_main.c    72%
  Function: singleshootStep                                97%
  Function: taskingStep                                   75%
  Function: taskingProcessUData                          95%
  Function: taskingProcessYData                          95%
  Function: pilaction                                    39%
  Function: main                                          80%
Module:  ..\..\fuelsys0_ert_rtw\fuelsys0.c                 42%
  Function: Sens_Failure_Counter                         13%
  Function: Fueling_Mode                                 16%
  Function: Init_controllogic                            100%
  Function: controllogic                                 48%
  Function: fuelsys0_step                                45%
  Function: fuelsys0_initialize                          100%
  Function: fuelsys0_terminate                           100%
Module:  MEMCPY_C                                         100%
Module:  MEMSET_C                                         100%
Module:  CPNNW                                           50%
Module:  MUL                                              0%
Module:  ..\..\slprj\ert\_sharedutils\
binarysearch_s16.c   89%
  Function: BINARYSEARCH_S16                             89%
Module:  ..\..\slprj\ert\_sharedutils\
dotproduct_s32s16.c  0%
  Function: DotProduct_s32s16                            0%
Module:  ..\..\slprj\ert\_sharedutils\
interpolate_even_s16_s16_sat.c  0%
  Function: INTERPOLATE_EVEN_S16_S16_SAT                 0%
Module:  ..\..\slprj\ert\_sharedutils\
interpolate_s16_s16_sat.c  56%
  Function: INTERPOLATE_S16_S16_SAT                     56%
Module:  ..\..\slprj\ert\_sharedutils\
look2d_s16_s16_s16_sat.c  100%
  Function: Look2D_S16_S16_S16_SAT                      100%
Module:  ..\..\slprj\ert\_sharedutils\
div_s32_sat_floor.c  67%
  Function: div_s32_sat_floor                            67%
Module:  ..\..\slprj\ert\_sharedutils\
fix2fix_s16_s32_sat.c  75%

```

Function: FIX2FIX_S16_S32_SAT	75%
Module: UDIL	29%
Module: UMOL	24%
Module: fuelsys0_ert_rtw_pil	0%
Module: CSTART	0%
Module: ..\..\fuelsys0_ert_rtw\fuelsys0_data.c	0%

pil_profiling_report =

Total Execution Time: 473348

	Cycles	%Cycles
Function: pilInitialize	18	0.004%
Function: initUDataProcessing	2828	0.597%
Function: processUData	1616	0.341%
Function: checkDataProcessingComplete	2020	0.427%
Function: pilStep	2626	0.555%
Function: initYDataProcessing	2828	0.597%
Function: processYData	1616	0.341%
Function: pilTerminate	16	0.003%
Function: getNextSymbol	37370	7.895%
Function: processData	61610	13.02%
Function: resetLibSymbolState	2828	0.597%
Function: checkDataProcessing	8080	1.707%
Function: singleshotStep	15150	3.201%
Function: taskingStep	1616	0.341%
Function: taskingProcessUData	9898	2.091%
Function: taskingProcessYData	9898	2.091%
Function: pilaction	5716	1.208%
Function: main	1886	0.398%
Function: Sens_Failure_Counter	3000	0.634%
Function: Fueling_Mode	8800	1.859%
Function: Init_controllogic	62	0.013%
Function: controllogic	17366	3.669%
Function: fuelsys0_step	66864	14.13%
Function: fuelsys0_initialize	54	0.011%
Function: fuelsys0_terminate	4	0.001%
Function: BINARYSEARCH_S16	41002	8.662%
Function: DotProduct_s32s16	0	0.000%
Function: INTERPOLATE_EVEN_S16_S16_SAT	0	0.000%
Function: INTERPOLATE_S16_S16_SAT	28678	6.059%

```

Function: Look2D_S16_S16_S16_SAT           32320 6.828%
Function: div_s32_sat_floor                 31782 6.714%
Function: FIX2FIX_S16_S32_SAT              4980 1.052%
Module:  MEMCPY_C                          23230 4.908%
Module:  MEMSET_C                           536 0.113%
Module:  CPNNW                              22624 4.780%
Module:  MUL                                0 0.000%
Module:  UDIL                               10624 2.244%
Module:  UMOL                               10292 2.174%
Module:  fuelsys0_ert_rtw_pil              0 0.000%
Module:  CSTART                             0 0.000%
147:    switch(tasking_pil_main_action) {

```

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**, 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 function, pilaction, 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: 3790326
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 2-11.

Tutorials

- | | |
|---|---|
| Tutorial: Using Option Sets (p. 5-2) | How to use option sets to switch between preconfigured project settings. |
| Tutorial: Creating New Template Projects (p. 5-4) | Steps for creating new template projects. |
| Tutorial: Configuring an Existing Model for Link for TASKING (p. 5-8) | An example showing how to configure an existing model for Link for TASKING. |
| Tutorial: Build Actions (p. 5-10) | How to use different build actions with Link for TASKING. |

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 Select **Start > Simulink > Link for TASKING > Select Preconfigured Target Preference Settings**.

The TASKING Configuration Selection dialog box appears.

- 2 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.

- 3 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 1-22. 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 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.

To create custom application and library template projects:

- 1** Select **Start > Simulink > Link for TASKING > Create New Template Projects**.
- 2** When prompted to select a configuration, 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 1-7).

- 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**.
- 3** When you are prompted, choose a location for the template projects, and enter the template name.
 - 4** When you are prompted, choose an option set. An *option set* delineates options specific to your target, such as whether you want to use simulator or hardware. You can use these to set up an initial configuration to modify later. See “Option Sets” on page 1-22 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. Your target preferences are automatically updated according to the option set you select. Messages at the command line inform you the following target preferences have changed:

- **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 to return to the default settings (using the **Start** menu option **Select Preconfigured Target Preference Settings**).

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

- 5** To modify the template projects, you need to open them in the TASKING EDE:
 - a** Select **Start > Simulink > Link for TASKING > Open Existing Template Projects**.
 - b** When you are prompted to select a configuration, select the same target for which you created new template projects, and click **OK**.

The template projects should now be open in the EDE.
 - c** Right-click the project in the TASKING EDE, and select **Project Options**. You can now modify the project options (compiler settings, linker settings, etc.).
 - d** When done, close the template projects in the TASKING EDE.
- 6** To modify your CrossView Pro configuration you need to specify a .ini file in the Initialization_File Target Preference field. See Initialization in the section “Target Preference Fields” on page 1-9.

You are now ready to use the configuration.

- 7** Open any Simulink model that is configured with Link for TASKING (tasking_demo_fuelsys, for example).
- 8** Select **Simulation > Configuration Parameters**. The Configuration Parameters dialog box opens.
- 9** Select **Link for TASKING** on the left-side panel. When you select your target in the **TASKING Configuration Description** menu, the template projects you have set up are used.

See “Template Projects” on page 2-5 for details about how Link for TASKING uses template projects during the build process.

You may want to create a new configuration to use with new template projects. See the next section, “Tutorial: Creating a New Configuration” on page 5-6 for details.

Tutorial: 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 **TASKING Configuration Description** menu. For example, if your target is TriCore, 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 **TASKING Configuration Description** menu in your model’s Configuration Parameters dialog box.

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

To create a new configuration:

- 1** From the MATLAB **Start** menu select **Simulink > Link for TASKING > TASKING Target Preferences**.

- 2 Select **Create new Configuration**, and click **OK**.
- 3 Expand **Configuration_Options**.
- 4 Type **Tutorial** in the **Configuration_Description** field.
- 5 Fill in the rest of the fields for this configuration. See “Setting Target Preferences” on page 1-7 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.

- 6 After you save your target preferences, you can use the new **Tutorial** configuration in any model that is configured with **Link for TASKING**. For example, open any of the **Link for TASKING** demo models (such as **tasking_demo_fuelsys**).
- 7 Select **Simulation > Configuration Parameters**. The **Configuration Parameters** dialog box opens.
- 8 Select **Link for TASKING** on the left-side panel. Click the **TASKING Configuration Description** menu, and notice that the **Tutorial** configuration now appears in the list.

Tutorial: Configuring an Existing Model for Link for TASKING

In this tutorial, you configure an existing fixed-point model and build it with Link for TASKING.

- 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 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 > Link for TASKING > Add Link for TASKING Configuration to Model** to add the Link for TASKING configuration set to the model.
- 4 Open the Configuration Parameters dialog box again from the **Simulation** menu, and verify that the Link for TASKING configuration set is now added to the model. Select **Link for TASKING** from the left panel:
 - a Set the **Build Action** to **Create and Build Application Project**.
 - b Select the **TASKING Configuration Description** to match your target.
 - c Select the check box option to **Specify Build Subdirectory Name**, and type `<target>_int` in the **Build Subdirectory Name** field. Replace the string `<target>` with your real target, e.g., `c166_int`.
 - 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 to avoid building the floating-point version of the `rtwlib` library.
 - e Under **Real-Time Workshop**, select **Hardware Implementation**, and select your device type. See the demo models for examples. Some devices use custom settings, others have preconfigured settings, 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.

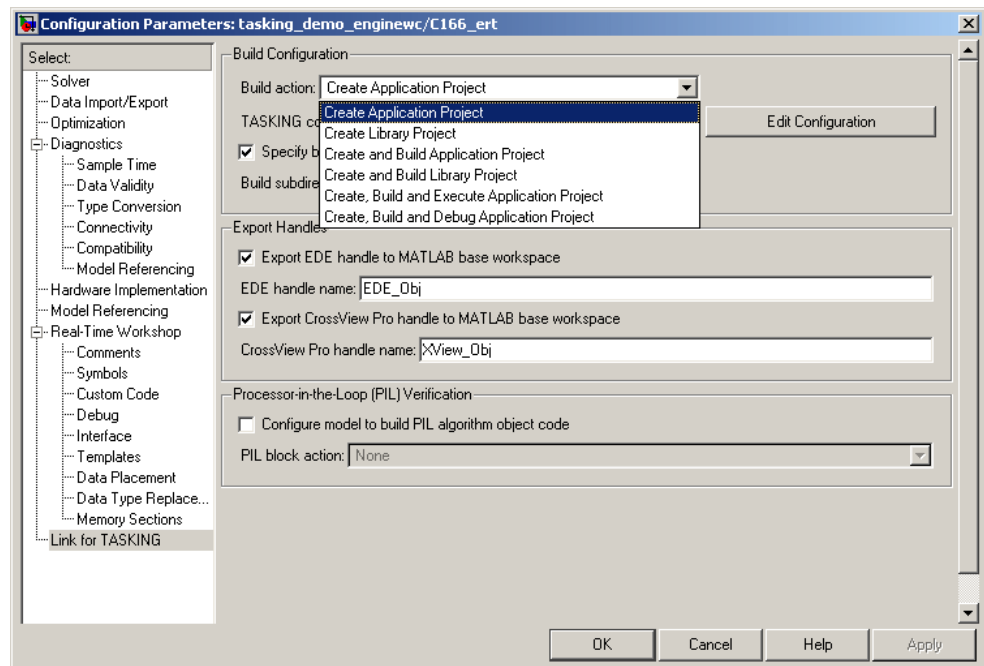
You are now ready to build the model. Press **Ctrl+B** or select **Tools > Real-Time Workshop > Build Model**.

Tutorial: Build Actions

Models configured with Link for TASKING have a build action setting that instructs Real-Time Workshop to perform different actions when the model is built. The following example explains what you can do by setting the build action.

Open any model configured with Link for TASKING (example, demo model `tasking_demo_fuelsys`).

Select **Simulation > Configuration Parameters**, and click the **Link for TASKING** configuration set. Under **Build Configuration**, the **Build Action** list has six different settings, as shown in the following figure.



- 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 DSP 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 this option is selected). This object allows you to interact with the TASKING EDE from MATLAB. For more information, see the section on using object handles, Chapter 3, “Objects”.

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 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.

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 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.

A CrossView Pro object handle is exported to the MATLAB workspace (if this option 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, Chapter 3, “Objects”.

- **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.

A

Add Link for TASKING Configuration to Model 1-20

B

build action 1-26
 setting 1-16
 tutorial 5-10
build configuration 1-26
build process
 command line information 2-11
 directory structure 2-10
 overview 2-2
 shared libraries 2-7
 template projects 2-5
build subdirectory name 1-26

C

classes 3-3
configuration options 1-24
configuration sets 1-13
Configure model to build PIL algorithm object code 1-27
Create a New Model (configured for Link for TASKING) 1-19
Create New Template Projects 1-19
CrossView Pro handle name 1-27

D

Demos 1-20

E

EDE handle name 1-26
Export CrossView Pro handle to MATLAB base workspace 1-26
Export EDE handle to MATLAB base workspace 1-26

Export handles 1-26

L

Launch and Test Communication with TASKING EDE 1-19
Link for TASKING
 build process 2-1
 introduction 1-2
 limitations and tips 1-28
 objects 3-1
 PIL cosimulation 4-1
 Start menu 1-18
 supported toolsets 1-4
 target preferences 1-7
 Tools menu 1-20
 tutorials 5-1
 user guide 1-6

M

methods
 tasking.edeapi 3-9
 tasking.edeproject 3-10
 tasking.edeprojectspace 3-10
 tasking.xviewapi 3-10

N

new configuration
 creating 5-6

O

objects
 accessing properties 3-7
 calling methods 3-7
 creating 3-4
 demo 3-7
 finding methods 3-6
 finding properties 3-7

- list of methods 3-9
- obtaining method help 3-6
- terms 3-2
- using 3-4

Open Existing Template Projects 1-20

- option sets 1-22
 - tutorial 5-2

Options 1-21

P

PIL block 4-7

- creating 4-5

PIL block action 1-27

PIL cosimulation

- building and downloading 4-10
- coverage and profiling reports 4-13
- debugging 4-11
- definitions 4-3
- how cosimulation works 4-4
- overview 4-2

Processor-in-the-Loop (PIL) Verification 1-27

project-based build process 2-4

R

Remove Link for TASKING Configuration
from Model 1-21

S

Select Preconfigured Target Preference
Settings 1-19

Specify build subdirectory name 1-26

T

target preferences

- fields 1-9
- setting 1-7

TASKING configuration description 1-26

TASKING CrossView Pro (Debugger)

- MATLAB API 1-3

TASKING EDE

- MATLAB API 1-3

TASKING Target Preferences

- Start menu 1-19
- Tools menu 1-20

template projects 2-5

- creating 5-4

tutorial

- build actions 5-10
- configuring existing models 5-8
- new configuration 5-6
- new template projects 5-4
- option sets 5-2

V

View, Modify, and Copy Configuration Sets via
Model Explorer 1-19