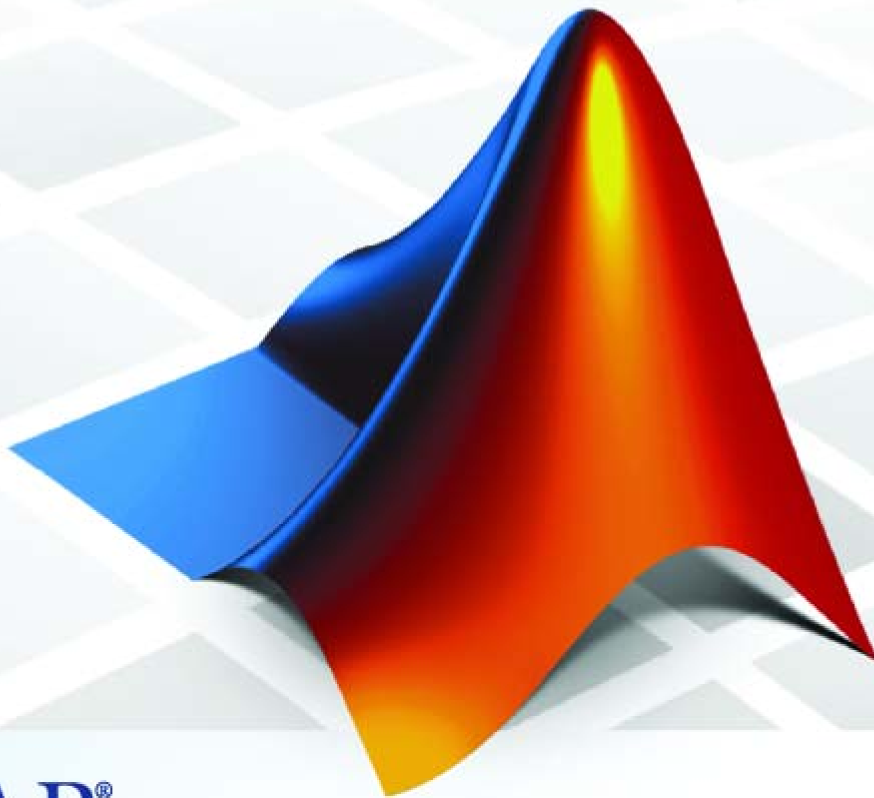
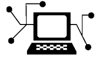
Embedded IDE Link™ MU 1
User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

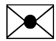


`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)



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

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

Embedded IDE Link™ MU User's Guide

© COPYRIGHT 2007 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, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

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

November 2007	Online only	New for Version 1.0 (Release 2007b+)

Contents

1 Getting Started

Product Overview **1-2**

The Structure and Components of Embedded IDE Link™ MU **1-4**
- Embedded IDE Link™ MU Components **1-4**
- Automation Interface **1-4**
- Project Generator **1-5**
- Verification **1-5**
- Configuring Embedded IDE Link™ MU and Green Hills® MULTI® Software **1-6**
- Configuring Green Hills® MULTI® to use Full Directory Paths **1-9**

2 Automation Interface

Getting Started with Automation Interface **2-2**
- Introducing the Automation Interface Tutorial **2-2**
- Starting and Stopping Green Hills MULTI® From the MATLAB® Desktop **2-5**
- Running the Interactive Tutorial **2-9**
- Querying Objects for Green Hills MULTI® Software **2-9**
- Loading Files into Green Hills MULTI® Software **2-11**
- Visibility and MULTI **2-12**
- Running the Project **2-13**
- Working With Data in Memory **2-13**
- More Memory Data Manipulation **2-16**
- Closing the Connections to Green Hills MULTI® Software **2-18**
- Tasks Performed During the Tutorial **2-19**

Constructing Objects **2-20**

Example — Constructor for ghsmulti Objects 2-20

Properties and Property Values 2-22
Working with Properties 2-22
Setting and Retrieving Property Values 2-22
Setting Property Values Directly at Construction 2-23
Setting Property Values with set 2-23
Retrieving Properties with get 2-24
Direct Property Referencing to Set and Get Values 2-24
Overloaded Functions for ghsmulti Objects 2-25

ghsmulti Object Properties 2-26
Quick Reference to ghsmulti Properties 2-26
Details About ghsmulti Object Properties 2-26

3 Project Generator

Introducing Project Generator 3-2

Using the Embedded IDE Link™ MU Blockset 3-3

Schedulers and Timing 3-9
Timer-Based Versus Asynchronous Interrupt Processing .. 3-9
Synchronous Scheduling 3-10
Asynchronous Scheduling 3-11
Scheduling Blocks 3-11
Asynchronous Scheduler Examples 3-12
Uses for Asynchronous Scheduling 3-15

Project Generator Tutorial 3-18
Process for Building and Generating a Project 3-18
Create the Model 3-19
Adding the Target Preferences Block to Your Model 3-20
Specifying Simulink® Configuration Parameters for Your Model 3-23
Creating Your Project 3-25

Setting Real-Time Workshop® Code Generation Options for Supported Processors **3-27**

Setting Real-Time Workshop® Category Options **3-30**
About Select Tree Category Options **3-30**
Target Selection **3-31**
Documentation **3-32**
Build Process **3-33**
Custom Storage Class **3-33**
Debug Pane Options **3-34**
Optimization Pane Options **3-35**
Embedded IDE Link™ MU Pane Options **3-37**
Overrun Indicator and Software-Based Timer **3-44**

Model Reference and Embedded IDE Link™ MU Software **3-45**
About Model Reference **3-45**
How Model Reference Works **3-45**
Using Model Reference with Embedded IDE Link™ MU Software **3-47**
Configuring Targets to Use Model Reference **3-48**

4 Verification

What Is Verification? **4-2**

Using Processor in the Loop **4-3**
Processor-in-the-Loop Overview **4-3**
PIL Block **4-6**
PIL Issues **4-6**
Creating and Using PIL Blocks **4-9**

Real-Time Execution Profiling **4-12**
Overview **4-12**
Profiling Program Execution **4-12**

5 Functions — By Category

Constructor 5-2

File and Project Operations 5-3

Processor Operations 5-4

Debug Operations 5-5

Data Manipulation 5-6

Status Operations 5-7

6 Functions — Alphabetical List

7 Blocks — By Category

Blackfin Support 7-2

Core Support 7-3

MPC5500 Support 7-4

Target Preferences 7-5

8 Blocks — Alphabetical List

A Examples

Automation Interface A-2

Working with Links A-2

Asynchronous Scheduler A-2

Project Generator A-2

Verification A-2

Index

Getting Started

Product Overview (p. 1-2)	Introduces Embedded IDE Link™ MU software
The Structure and Components of Embedded IDE Link™ MU (p. 1-4)	Describes the two components of Embedded IDE Link™ MU

Product Overview

Embedded IDE Link™ MU software provides an interface between MATLAB® and the Green Hills MULTI® IDE software. The software enables you to

- Access the processor
- Manipulate data on the processor
- Manage projects within the IDE

while using the MATLAB® numerical analysis and simulation functions.

Embedded IDE Link™ MU connects MATLAB® and Simulink® with Green Hills MULTI® integrated development and debugging environment from Green Hills Software (GHS). The software enables you to use MATLAB® and Simulink® to debug and verify embedded code running on many microprocessors that Green Hills MULTI® software supports, such as the Freescale™ MPC5500 and MPC7400, Blackfin®, and NEC® V850 families.

Using Embedded IDE Link™ MU software, you can perform the following tasks and others related to Model-Based Design:

- Function calls — Write scripts in MATLAB® to execute any function in the Green Hills MULTI® IDE
- Automation — Write automated tests in MATLAB® to execute on your processor, including control and verification operations
- Host-Processor Communication — Communicate with the processor directly from MATLAB®, without going to the IDE
- Verification and Validation
  - Load and execute projects into the Green Hills MULTI® IDE software from the MATLAB® command line
  - Build and compile code, and then use vectors of test data and parameters to test the code
  - Build and compile your code, and then download the code to the processor and execute it

- Design models — Design models and algorithms in MATLAB® and Simulink® and run them on the processor
- Generate code — Generate executable code for your processor directly from the models designed in Simulink®, and execute it

Embedded IDE Link™ MU software includes a project generator component. With the project generator component, you can generate a complete project file for Green Hills MULTI® software from Simulink® models, using C code generated with Real-Time Workshop® software. Thus, you can use both Real-Time Workshop® and Real-Time Workshop® Embedded Coder™ software to generate generic ANSI C code projects for Green Hills MULTI® from Simulink® models. You can then build and run the code on supported processors.

The following list suggests some of the uses for Embedded IDE Link™ MU:

- Create test benches in MATLAB® and Simulink® for testing your manually written or automatically generated code running on a variety of DSPs
- Generate code and project files for Green Hills MULTI® software from Simulink® models using both Real-Time Workshop® and Real-Time Workshop® Embedded Coder™ software for rapid prototyping or deployment of a system or application
- Build, debug, and verify embedded code on supported processors with MATLAB®, Simulink®, and Green Hills MULTI® software
- Perform processor-in-the-loop (PIL) testing of embedded code

The Structure and Components of Embedded IDE Link™ MU

In this section...

"Embedded IDE Link™ MU Components" on page 1-4

"Automation Interface" on page 1-4

"Project Generator" on page 1-5

"Verification" on page 1-5

"Configuring Embedded IDE Link™ MU and Green Hills® MULTI® Software" on page 1-6

"Configuring Green Hills® MULTI® to use Full Directory Paths" on page 1-9

Embedded IDE Link™ MU Components

Embedded IDE Link™ MU software comprises these components

- Automation Interface — Enables communication between MATLAB® and Green Hills® MULTI® software.
- Project Generation — Uses Simulink® to let you build models, simulate them, and generate code from the models directly to the processor.
- Verification — Validate and verify your projects. You can simulate algorithms and processes in Simulink® models and concurrently on your processor. Comparing the concurrent simulation results helps verify the fidelity of your model or algorithm code.

Automation Interface

The Automation Interface component enables you to use MATLAB® functions and methods to communicate with the Green Hills MULTI® IDE software. With the MATLAB® functions, you can perform the following program development tasks:

- Automate project management.
- Debug projects by manipulating the data in the processor memory (internal and external) and registers.
- Exercise functions from your project on the processor.

- Communicate between the host and processor applications.

The Automation Interface component provides the following types of functionality:

- Debug Component — Methods and functions for project automation, debugging, and data manipulation.
- Function Call Component — Methods that enable you to invoke individual functions on the processor.
- Host Processor Communication Component — Methods that support various standard communication protocols, such as BTC, TCP/IP, and UDP.

Project Generator

The Project Generator component is a collection of methods that use the Green Hills MULTI® API to create projects in Green Hills MULTI® and generate code with Real-Time Workshop®. With the interface, you can do the following:

- Automatic project-based build process — Automatically create and build projects for code generated by Real-Time Workshop® or Real-Time Workshop® Embedded Coder™.
- Custom code generation — Use Embedded IDE Link™ MU with any Real-Time Workshop® Embedded Coder™ System Target File (STF) to generate both processor-specific and optimized code.
- Automatic downloading and debugging — Debug generated code in the Green Hills MULTI® debugger, using either the instruction set simulator or real hardware.
- Create and build projects for Green Hills MULTI® from Simulink® models — Project Generator uses Real-Time Workshop® or Real-Time Workshop® Embedded Coder™ to build projects that work with supported processors.
- Generate custom code using the Configuration Parameters in your model with the system target files `multilink_ert.tlc` and `multilink_grt.tlc`.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded IDE Link™ MU provide the following verification tools.

- **Processor in the loop (PIL) cosimulation** — Use cosimulation techniques to verify generated code running in an instruction set simulator or real hardware environment.
- **Execution profiling** — Gather execution profiling measurements with Green Hills MULTI® instruction set simulator to establish the timing requirements of your algorithm.

Configuring Embedded IDE Link™ MU and Green Hills® MULTI® Software

Embedded IDE Link™ MU requires some information about your MULTI installation before you can use the software to develop projects in MULTI from MATLAB®. To configure the interface between MATLAB®and MULTI, provide the information in the following table. Embedded IDE Link™ MU provides a GUI-based configuration utility to help you configure the software and interface.

GUI Parameter	Configuration Information	Description
Directory	MULTI installation directory	Identifies the path to your Green Hills® software.
Configuration	Primary processor	Identifies the processor on which you are developing.
Debug server	Debug server type	Specifies the type of debug server to use.
Host name	Host name	Specifies the name of the machine that runs your Embedded IDE Link™ MU service.
Port number	Port number	Specifies the port for communicating with the host and Embedded IDE Link™ MU service. The service listens on this port.

Configuring Your Embedded IDE Link™ MU Software

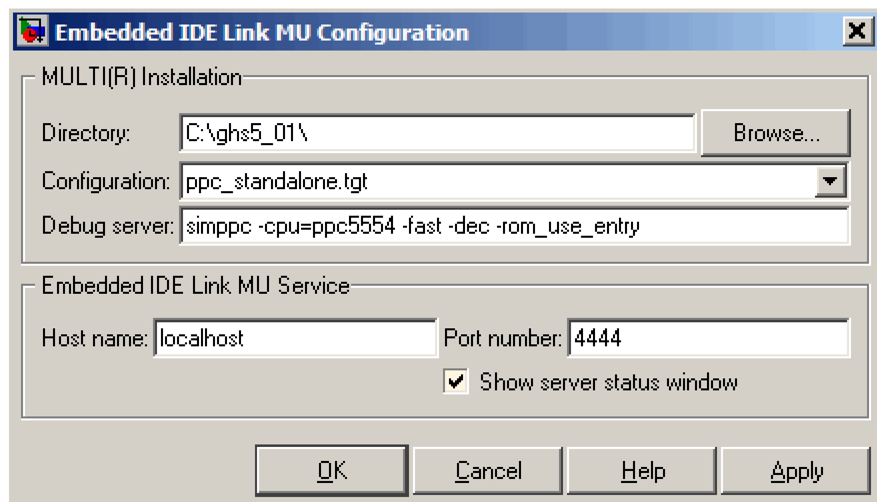
You must configure your installation before you start working with the software and MULTI. Follow these steps to open the Embedded IDE Link™ MU configuration utility.

Note You must perform this configuration process before using Embedded IDE Link™ MU software.

---

1 Enter `ghsmulticonfig` at the MATLAB® prompt.

The Embedded IDE Link™ MU Configuration dialog box opens, as shown in the following figure.



2 In the **Directory** field, enter the path to the executable file `multi.exe` for your Green Hills MULTI installation. Click **Browse** to search for the file if necessary.

3 From the **Configuration** list, select your primary processor. Embedded IDE Link™ MU supports a variety of processors. Choose one that matches your development platform. In many cases, the *`processor`*`_standalone.tgt` variants, such as `ppc_standalone.tgt`, work

well. Refer to your Green Hills MULTI documentation for more information about the configuration options for processors.

4 Enter the debug server string in **Debug server**. The string you enter sets specific values for processors, such as the board support library and whether the processor is big or little endian.

   The standard input string is `debugconnection`. To use a processor simulator, such as the MPC5554 simulator, enter the string

   ```
   simppc -cpu=ppc5554 -fast -dec-rom_use_entry
   ```

   as shown in the figure. Your MULTI documentation provides more information about the debug server options and how to use them. You can find more debug server string for simulators in the reference material for `ghsmulticonfig`.

5 In **Host name**, enter the name of the machine that is going to run the Embedded IDE Link™ MU service. When you construct a `ghsmulti` object, the `ghsmulti` function starts the Embedded IDE Link™ MU service. To launch the service, the function needs to know where the service will run. The **Host name** string identifies that location. The default value is `localhost`, meaning the service runs on the local machine. No other input is valid.

6 Enter the port number for the service in **Port number**.

   Port number 4444 is the default port value. To change the port used, enter a different value in this field. Verify that the port you enter is available. If the port number you enter is not available, the Embedded IDE Link™ MU service does not start. Thus, you get an error message in MATLAB® when you try to construct a `ghsmulti` object.

7 Select or clear **Show server status window** to specify whether the Embedded IDE Link™ MU service status appears in the task bar. The default value is to show the service status. Clearing **Show server status window** hides the status in the task bar. Select this option as a best practice. Keeping this option selected enables the software to shut down the communication services for Green Hills® MULTI® completely.

8 Click **OK** to complete the configuration process and close the dialog box.

Configuring Green Hills® MULTI® to use Full Directory Paths

When you install MULTI to use with the software, MULTI sets the **Show Paths** option to use relative file paths. To ensure that projects and programs build correctly, configure MULTI to use full directory paths. Follow these steps to change the configuration in MULTI.

1. Start MULTI from your desktop.
2. Switch to the Project Manager tool.
3. Select **View > Show Paths > Full Paths**.

Automation Interface

Getting Started with Automation Interface (p. 2-2)	Guides you through the process of creating and using objects and embedded objects
Constructing Objects (p. 2-20)	Explains what a `ghsmulti` object is and how to construct one
Properties and Property Values (p. 2-22)	Describes how to work with objects, their properties and property values
ghsmulti Object Properties (p. 2-26)	Describes the properties of `ghsmulti` objects

Getting Started with Automation Interface

In this section...

"Introducing the Automation Interface Tutorial" on page 2-2

"Starting and Stopping Green Hills MULTI® From the MATLAB® Desktop" on page 2-5

"Running the Interactive Tutorial" on page 2-9

"Querying Objects for Green Hills MULTI® Software" on page 2-9

"Loading Files into Green Hills MULTI® Software" on page 2-11

"Visibility and MULTI" on page 2-12

"Running the Project" on page 2-13

"Working With Data in Memory" on page 2-13

"More Memory Data Manipulation" on page 2-16

"Closing the Connections to Green Hills MULTI® Software" on page 2-18

"Tasks Performed During the Tutorial" on page 2-19

Introducing the Automation Interface Tutorial

Embedded IDE Link™ MU provides a connection between MATLAB® software and a processor in Green Hills MULTI® development environment. You use MATLAB® objects as a mechanism to control and manipulate a signal processing application using the computational power of MATLAB® software. This approach can help you while you debug and develop your application. Another possible use for automation is creating MATLAB® scripts that verify and test algorithms that run in their final implementation on your production processor.

Note Before using the functions available with the objects, you must designate a server and processor in Green Hills MULTI® software. The object you create is specific to the server and processor you specify.

To help you start using objects in the software, Embedded IDE Link™ MU software includes a tutorial—`multilinkautointtutorial.m`. As you work through this tutorial, you perform the following tasks that step you through creating and using objects to interact with the Green Hills MULTI® IDE:

1. Select your primary server and port.
2. Create and query objects to Green Hills MULTI® IDE.
3. Use MATLAB® to load files into Green Hills MULTI® IDE.
4. Work with your Green Hills MULTI® IDE project from MATLAB®.
5. Close the connections you opened to Green Hills MULTI® IDE.

The tutorial covers some methods and functions for the software. The following tables show functions and methods for the software. The functions do not require an object. The methods require an existing `ghsmulti` object to use as an input argument for the method.

Functions for Working with Green Hills MULTI®

The following table shows functions that do not require an object.

Function	Description
`ghsmulti`	Construct an object that refers to a Green Hills MULTI® IDE instance. When you construct the object you specify the IDE instance by host and port.
`ghsmulticonfig`	Set Embedded IDE Link™ MU software preferences.

Methods for Working with ghsmulti Objects in Green Hills MULTI®

The following table presents some of the methods that require a `ghsmulti` object.

Methods	Description
`add`	Add file to project
`address`	Return address and page for entry in symbol table in Green Hills MULTI® IDE
`build`	Build project in Green Hills MULTI®
`cd`	Change working directory
`connect`	Connect IDE to processor
`display`	Display properties of object that references Green Hills MULTI® IDE
`halt`	Terminate execution of process running on processor
`isrunning`	Test whether processor is executing process
`load`	Load built project to processor
`open`	Open file in project
`read`	Retrieve data from memory on processor
`regread`	Read values from processor registers
`regwrite`	Write data values to registers on processor
`reset`	Restore program counter (PC) to entry point for current program.
`restart`	Restore processor to program entry point
`run`	Execute program loaded on processor
`save`	Save files or projects.
`visible`	Set whether Green Hills MULTI® IDE window is visible on desktop while Green Hills MULTI® IDE is running
`write`	Write data to memory on processor

Running Green Hills MULTI® on Your Desktop — Visibility

When you create a `ghsmulti` object in the tutorial in the next section, Embedded IDE Link™ MU starts Green Hills MULTI®in the background.

If Green Hills MULTI® is running in the background, the IDE windows, such as the editor and debugger, do not appear on your desktop. MULTI does appear in your task bar and on the **Applications** page in the Task Manager. It shows up as a process, `IDE.exe`, on the **Processes** tab in Task Manager.

You can make the Green Hills MULTI® IDE visible with the function `visible`. To close the IDE when it is not visible and MATLAB® is not running, use the **Processes** tab in Windows Task Manager and look for `IDE.exe`.

If an object that refers to Green Hills MULTI® exists when you close Green Hills MULTI®, the application does not close. Windows moves it to the background (it becomes invisible). Only after you clear all objects that access Green Hills MULTI®, or close MATLAB, does closing Green Hills MULTI® unload the application. You can see if Green Hills MULTI® is running in the background by checking in the Windows Task Manager or the task bar. When Green Hills MULTI® is running, the entry `IDE.exe` appears in the **Image Name** list on the **Processes** tab.

Starting and Stopping Green Hills MULTI® From the MATLAB® Desktop

Embedded IDE Link™ MU software provides you the ability to control MULTI software from the MATLAB® command window. When you create a `ghsmulti` object, MATLAB® starts the services shown in the following table to enable MATLAB® to communicate with the Green Hills® MULTI® IDE:

Service Type	Process Name	Description
Python Service	`mpythonrun.exe`	Python is a programming language the software uses to establish a connection between MATLAB® and MULTI.
Python Service	`svc_python.exe`	Connection to IDE.
Python Service	`svc_router.exe`	Connection to IDE.
Python Service	`svc_statemgr.exe`	Connection to IDE

Service Type	Process Name	Description
Python Service	`svc_window.exe`	Connection to IDE.
Embedded IDE Link™ MU service	Not applicable	Enables MATLAB® to send commands to the Green Hills® MULTI® development environment. This is a child process of the python services.

Each time you create a `ghsmulti` object, the software starts another set of the python services shown in the table.

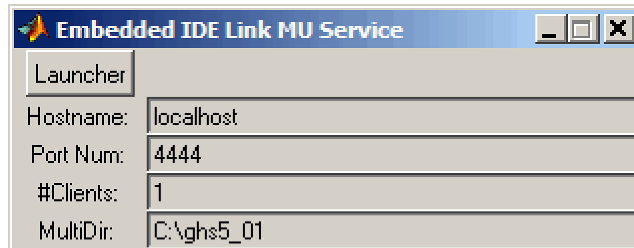
Starting Green Hills® MULTI® From MATLAB®

When you use the `ghsmulti` function, the software starts two classes of services—python services and the Embedded IDE Link™ MU service. The entries in the following table describe how the software controls the IDE when you create a `ghsmulti` object:

Create ghsmulti Object with ghsmulti Function	Status of IDE	Result
`id=ghsmulti`	Not running	The software starts the Embedded IDE Link™ MU service and the IDE and connects to the default host name and port number—`localhost` and `4444`.
`id=ghsmulti('hostname','localhost','portnum',4444)`	Not running	The software starts the Embedded IDE Link™ MU service and the IDE and connects to the specified host name and port number—`localhost` and `4444`.

Create ghsmulti Object with ghsmulti Function	Status of IDE	Result
`id2=ghsmulti`	Running	The software connects to the existing Embedded IDE Link™ MU service connected to the default host name and port.
`id2=ghsmulti('hostname','localhost','portnum',4446)`	Running	The software starts a new Embedded IDE Link™ MU service connected to the specified host name and port number.

When the software starts the Embedded IDE Link™ MU service, the following service dialog box appears on your desktop:



Information in the window provides details about the service. Clicking **Launcher** opens the MULTI Launcher utility.

Stopping Green Hills® MULTI® From MATLAB®

After you complete your development work with the software, best practice suggests that you close the IDE from MATLAB®. Two conditions control how you close the IDE, as shown in the following table:

Embedded IDE Link™ MU Service State	To Close the IDE
One or more services appear in the task bar and Embedded IDE Link™ MU service dialog boxes are visible.	Perform these steps: **1** Enter `clear all` in MATLAB® to remove the `ghsmulti` objects from your workspace. **2** Verify that the MULTI clients are no longer connected by checking that **#Clients** in each service dialog box is `0`. **3** Close the service dialog boxes.
Services appear in the task bar but the service dialog boxes are not visible.	Perform these steps: **1** Enter `clear all` inMATLAB® to remove the `ghsmulti` objects from your workspace. **2** Open the Microsoft®Windows Task Manager. **3** Click **Processes**. **4** Select `svc_router.exe` from the list. Closing this service stops `mpythonrun.exe`, `svc_window.exe`, and `svc_statemgr.exe`. **5** Click **End Now**. **6** Select `svc_python.exe` from the list. **7** Click **End Now**.

Note Clicking the task bar icon for the service and selecting close does not close the IDE correctly.

Running the Interactive Tutorial

You have the option of running this tutorial from the MATLAB® command line or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB®, click `run multilinkautointtutorial`. This command launches the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next section. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial M-file used here by clicking `multilinkautointtutorial.m`.

Querying Objects for Green Hills MULTI® Software

In this tutorial section you create the connection between MATLAB® and Green Hills MULTI® IDE. This connection, or `ghsmulti` object, is a MATLAB® object that you save as variable `id`. You use function `ghsmulti` to create `ghsmulti` objects. `ghsmulti` supports input arguments that let you specify values for `ghsmulti` object properties, such as the global timeout. Refer to the `ghsmulti` reference information for more about the input arguments.

Use the generated object `id` to direct actions to your project and processor. In the following tasks, `id` appears in all method syntax that interact with the IDE primary target and the processor: The object `id` identifies and refers to a specific instance of the IDE.

You must include the object in any method syntax you use to access and manipulate a project or files in a session in Green Hills MULTI® software:

1. Create an object that refers to your selected service and port. Enter the following command at the prompt.

   ```
   id = ghsmulti('hostname','localhost','portnum',4444)
   ```

   If you watch closely, and your machine is not too fast, you see Green Hills MULTI® appear briefly when you call `ghsmulti`. If Green Hills MULTI®

was not running before you created the new object, Green Hills MULTI® launches and runs in the background.

Usually, you need to interact with Green Hills MULTI® while you develop your application. The function `visible`, controls the state of Green Hills MULTI® on your desktop. `visible` accepts Boolean inputs that make Green Hills MULTI® either visible on your desktop (input to `visible` ≥ 1) or invisible on your desktop (input to `visible` = 0). For this tutorial, you need to interact with the development environment, so use `visible` to set the IDE visibility to `1`.

2. To make Green Hills MULTI® show on your desktop, enter the following command at the command prompt:

   ```
   visible(id,1)
   ```

3. Next, enter `display(id)` at the prompt to see the status information.

   ```
   MULTI Object:
     Host Name        : localhost
     Port Num         : 4444
     Default timeout  : 10.00 secs
     MULTI Dir        : C:\ghs\multi500\ppc\
   ```

   Embedded IDE Link™ MU provides three methods to read the status of a processor:

   - `info` — Return a structure of testable session conditions.
   - `display` — Print information about the session and processor.
   - `isrunning` — Return the state (running or halted) of the processor.

4. Verify that the processor is running by entering

   ```
   runstatus = isrunning(id)
   ```

   The MATLAB® prompt responds with message that indicates the processor is stopped:

   ```
   runstatus =

         0
   ```

Loading Files into Green Hills MULTI® Software

You have established the connection to a processor and board. Using three methods you learned about the hardware, whether it was running, its type, and whether Green Hills MULTI® IDE was visible. Next, give the processor something to do.

In this part of the tutorial, you load the executable code for the CPU in the IDE. Embedded IDE Link™ MU includes a tutorial project file for Green Hills MULTI®. Through the next commands in the tutorial, you locate the tutorial project file and load it into Green Hills MULTI®. The `open` method directs Green Hills MULTI® to load a project file or workspace file.

Note To continue the tutorial, you must identify or create a directory to which you have write access. Embedded IDE Link™ MU cannot create a directory for you. Create one in the Microsoft® Windows directory structure before you proceed with the this tutorial.

Green Hills MULTI® has its own workspace and workspace files that are quite different from MATLAB® workspace files and the MATLAB® workspace. Remember to monitor both workspaces. To change the working directory to your writable directory:

1. Use `cd` to switch to the writable directory

   ```
   prj_dir=cd('C:\ide_link_mu_demo')
   ```

   where the name and path to the writable directory is a string, such as `C:\ide_link_mu_demo` as used in the example. Replace `C:\ide_link_mu_demo` with the full path to your writable directory.

2. Change your working directory to the new directory by entering the following command:

   ```
   cd(id,prj_dir)
   ```

3. Use the following command to create a new Green Hills MULTI® project named `debug_demo.gpj` in the new directory:

```
new(id,'debug_demo.gpj')
```

Switch to the IDE to verify that your new project exists. Next, add source files to your project.

4 Add the provided source file—`multilinkautointtutorial.c` to the project `debug_demo.gpj` using the following command:

```
add(id,'multilinkautointtutorial')
```

5 Save your project.

```
save(id,'my_debug_demo.gpj','project')
```

Your IDE project is saved with the name `my_debug_demo.gpj` in your writable directory. The input string '`project`' specifies that you are saving a project file.

6 Next, set the build options for your project. Use the following command to set the compiler build options to use and specify a processor (optional).

```
setbuildopt(id,'Compiler','-G',-cpu=V850)
```

The input argument `-cpu=V850` is optional to specify the processor. Change to processor designation to match your processor if necessary.

Visibility and MULTI

If MULTI is not running on your desktop when you create the `multilink` object, Embedded IDE Link™ MU starts MULTI and then configures it to run in the background. Verify that MULTI is running by checking that MULTI appears on your task bar

Usually you need to interact with the IDE, so Embedded IDE Link™ MU provides a function called `visible` that controls whether MULTI is visible. `visible` takes the following Boolean input argument:

- `0` hides the IDE on your desktop. It appears on the task bar.
- `1` makes all components of the IDE visible on your desktop.

The remainder of this tutorial requires that you interact with the IDE.

```
visible(id,1)  % Make the IDE visible on the desktop.
```

Running the Project

After you create dot_project_c.gpj in the IDE, you can use Embedded IDE Link™ MU functions to create executable code from the project and load the code to the processor.

To build the executable and download and run it on your processor:

1. Use the following build command to build an executable module from the project debug_demo.gpj.

   ```
   build(id,'all',20) % The optional input argument 20 sets the time-out period to 20 seconds.
   ```

2. To load the new executable to the processor, use load with the project file name and the object name. The name of the executable is debug_demo.

   ```
   load(id,'debug_demo',30); % Set time-out value to 30 seconds.
   ```

Embedded IDE Link™ MU provides methods to control processor execution—run, halt, and reset. To demonstrate these methods, use run to start the program you just loaded on to the processor, and then use halt to stop the processor.

1. Enter the following methods at the command prompt and review the response in the MATLAB® command window.

   ```
   run(id)      % Start the program running on the processor.
   halt(id)     % Halt the processor.
   reset(id)    % Reset the program counter to start of program.
   ```

   Use isrunning after the run method to verify that the processor is running. After you stop the processor, isrunning can verify that the processor has stopped.

Working With Data in Memory

Embedded IDE Link™ MU provides methods that enable you to read and write data to memory on the processor. Reading and writing data depends on the symbol table for your project. The symbol table is available only after you load the executable into the debugger. This sections introduces address

and dec2hex. Use them to read the addresses of two global variables—ddat and idat.

1 After you load debug_demo into the debugger, enter the following commands to read the addresses of ddat and idat:

```
ddatA=address(id,'ddat')
ddatA =
     3145744        0

ddatI=address(id,'idat')

ddatI =

     3145728           0
```

2 Review the results in hexadecimal representation.

```
dec2hex(ddatA)

ans =

300010
000000

dec2hex(ddatI)

ans =

300000
000000
```

After you load the target code to the processor, you can examine and modify data values in memory, as the previous read function examples demonstrated.

For non-changing data values in memory (static values), the values are available immediately after you load the program file.

A more interesting case is looking at variable values that change during program execution. Manipulating changing data values at intermediate points during execution can provide helpful analysis and verification information.

To enable you to read and write data while your program is running, the software provides methods to insert and delete breakpoints in the source programs. Inserting breakpoints lets you pause program execution to read or change variable data values. You cannot change values while your program is running.

The method `insert` creates a new breakpoint at either a source file locations, such as a line number, or at a physical memory address. `insert` takes either the line number or the address an as input argument.

To read the values in the next section of this tutorial, use the following methods to insert breakpoints at lines 24 and 29 in the source file `multilinkautointtutorial.c`

1 Change directories to your original working directory.

```
cd(id,proj_dir);
```

2 (Optional for convenience) Create variables for the line numbers in the source file.

```
brkpt24 = 24;
brtpt29 = 29;
```

3 Use the following commands to insert breakpoints on line 24 and line 29 of the source file:

```
insert(id,'multilinkautointtutorial',brkpt24); % Insert breakpoint on line 24.
insert(id,'multilinkautointtutorial',brkpt29); % Insert breakpoint on line 29.
```

4 Open and activate the file in the IDE from the MATLAB® command window by issuing the following commands:

```
open(id,'multilinkautointtutorial');
activate(id,'multilinkautointtutorial');
```

Activating multilinkautointtutorial.c transfers focus in the IDE to the activated file. Switch to the IDE to verify that the file is in your project and open.

When you look in the IDE debugger window, the breakpoints you added to multilinkautointtutorial.c are marked by a STOP sign icon on lines 24 and 29.

A similar method, remove, deletes breakpoints.

To help you inspect the source file in the IDE and verify the breakpoints, the open and activate methods display the file multilinkautointtutorial.c in the IDE and force the source file to the front.

One final method actually connects the IDE to your hardware or simulator. connect takes a ghsmulti object as an input argument to connect the specific IDE primary target referenced by id to the associated processor.

More Memory Data Manipulation

The source file multilinkaautointtutorial.c defines two 1-by-4 global data arrays—ddat and idat. You can locate the declaration in the file. Embedded IDE Link™ MU software provides the read and write methods so you can access the arrays from MATLAB®. Find the declaration and note the initialization values.

This tutorial section demonstrates reading and writing data in memory, and controlling the processor.

1 Get the address of the symbols ddat and idat. Enter the following commands at the prompt.

```
ddat_addr=address(id,'ddat'); % Get address from symbol table.
idat_addr=address(id,'idat');
```

2 Create two MATLAB® variables to specify the data types for ddat and idat.

```
ddat_type-'double';
idat_type='int32';
```

3 Declare some values in two MATLAB® variables.

```
ddat_value=double([pi 12.3 exp(-1) sin(pi/4)]);
idat_value=int32(1:4);
```

4 Stop the processor.

```
halt(id)
```

5 Reload the project. If you did not save the source file in the project, reloading the project removes the breakpoints you added and move the program counter (PC) to the start of the program.

```
reload(id,100); % Reload the program file (.gpj) and reset the PC to p
```

6 Use the following commands to restore the breakpoints on line 24 and 29.

```
insert(id,'multilinkautointtutorial.c',brkpt24);
insert(id,'multilinkautointtutorial.c',brkpt29);
```

7 Use the following method to connect the IDE to the processor:

```
connect(id);
```

8 With the breakpoints in the code, run the program until it stops at the first breakpoint on line 24.

```
run(id,'runtohalt',30); % Change the time-out to 30 seconds to let the
```

9 Check the current values stored in `ddat` and `idat`. Later in this tutorial you change these values from MATLAB®.

```
ddatV=read(id,address(id,'ddat',ddat_type,4) % Should match the initial
idatV=read(id,address(id,'idat',idat_type,4)
```

MATLAB® displays the values of `ddatV` and `idatV`.

```
ddatV=

   16.300   -2.1300   5.1000   11.8000

idatV=

         1 508    646    7000
```

10 Change the values in ddat and idat by writing new values to the memory addresses.

```
write(id,address(id,'ddata'),ddat_value) % Write pi, 12.3, exp(-1), and
write(id,address(id,'idat'),idat_value) % Write  the vector [1:4] to me
```

11 Resume the program execution from the breakpoint and run until the program stops.

```
run(id,'runtohalt','30); % Stop at next breakpoint (line 29).
```

12 Read the values in memory for ddat and idat to verify the changes.

```
ddatV = read(id,address(id(id,'ddat'),ddat_type,4) % Read the data as double data type.

ddatV=

3.1416   12.3000   0.3679   0.7071

idatV = read(id,address(id,'idat'),idat_type,4) % Read the data as int32 data type.

idatV=

        1    2    3    4
```

The data stored in ddat and idat are what you wrote to memory.

13 After you review the data, restart the processor to run to return the PC to the program start.

```
restart(id);
```

Closing the Connections to Green Hills MULTI® Software

Objects that you create in Embedded IDE Link™ MU software have connections to Green Hills MULTI® IDE. Until you delete these objects, the Green Hills MULTI® process (Idde.exe in the Windows Task Manager) remains in memory. Closing MATLAB® removes these objects automatically, but there may be times when it helps to delete the handles manually, without quitting MATLAB.

Note When you clear the last `ghsmulti` object, the software does not close the running Embedded IDE Link™ MU service. When it closes the IDE, it does not save current projects or files in the IDE, and it does not prompt you to save them. A best practice is to save all of your projects and files before you clear `ghsmulti` objects from your MATLAB® workspace.

Use the following commands to close the project files in Green Hills MULTI® IDE and remove the breakpoints you added to the source file.

```
close(id,'debug_demo.gpj','project') % Close the project file.
visible(id,1) % Make MULTI visible.
remove(id,'multilinkautointtutorial.c',brkpt24);

remove(id,'multilinkautointtutorial.c',brkpt29);
```

Finally, to delete your link to Green Hills MULTI® use `clear id`.

You have completed the Automation Interface tutorial using Embedded IDE Link™ MU.

Tasks Performed During the Tutorial

During the tutorial you performed the following tasks:

1. Created and queried objects that refer to a session in Embedded IDE Link™ MU to get information about the session and processor.

2. Used MATLAB® software to load files into the Green Hills MULTI® IDE and used methods in MATLAB software to run that file.

3. Closed the links you opened to Green Hills MULTI® software.

This set of tasks is used in any development work you do with signal processing applications. Thus, the tutorial gives you a working process for using Embedded IDE Link™ MU and your signal processing programs to develop programs for a range of processors.

Constructing Objects

When you create a connection to a session in Green Hills MULTI® using the `ghsmulti` function, you create a `ghsmulti` object (in object-oriented design terms, you *instantiate* the `ghsmulti` object). The object implementation relies on MATLAB® object-oriented programming capabilities like the objects in MATLAB® or Filter Design Toolbox™ software.

The discussions in this section apply to the objects in Embedded IDE Link™ MU. Because `ghsmulti` objects use the MATLAB software techniques, the information about working with the objects, such as how you get or set object properties or use methods, apply to the `ghsmulti` objects in Embedded IDE Link™ MU.

Like other MATLAB® structures, ghsmulti objects have predefined fields referred to as *object properties*.

You specify object property values by the following methods:

- Specifying the property values when you create the object
- Creating an object with default property values, and changing some or all of these property values later

For examples of setting link properties, refer to "Setting Property Values with set".

Example — Constructor for ghsmulti Objects

The easiest way to create an object is to use the function `ghsmulti` to create an object with the default properties. Create an object named `id` referring to a session in Green Hills MULTI® by entering the following syntax:

```
id = ghsmulti
```

MATLAB® responds with a list of the properties of the object `id` you created along with the associated default property values.

```
MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
```

```
Default timeout  : 10.00 secs
MULTI Dir        : C:\ghs\multi500\ppc\
```

The object properties are described in the `ghsmulti` documentation.

Note These properties are set to default values when you construct links.

Properties and Property Values

In this section...

"Working with Properties" on page 2-22

"Setting and Retrieving Property Values" on page 2-22

"Setting Property Values Directly at Construction" on page 2-23

"Setting Property Values with set" on page 2-23

"Retrieving Properties with get" on page 2-24

"Direct Property Referencing to Set and Get Values" on page 2-24

"Overloaded Functions for ghsmulti Objects" on page 2-25

Working with Properties

Links (or objects) in this Embedded IDE Link™ MU software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. Also, a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

Setting and Retrieving Property Values

You can set `ghsmulti` object property values by either of the following methods:

- Directly when you create the link — see "Setting Property Values Directly at Construction"
- By using the `set` function with an existing link — see "Setting Property Values with set"

Retrieve `ghsmulti` object property values with the `get` function.

Direct property referencing lets you either set or retrieve property values for `ghsmulti` objects.

Setting Property Values Directly at Construction

To set property values directly when you construct an object, include the following entries in the input argument list for the constructor method `ghsmulti`:

- A string for the property name to set, followed by a comma. Enclose the string in single quotation marks as you do any string in MATLAB®.
- The property value to associate with the named property. Sometimes this value is also a string.

You can include as many property names in the argument list for the object construction command as there are properties to set directly.

Example — Setting Link Property Values at Construction

Create a connection to an instance of the IDE in Green Hills MULTI® software and set the following object properties:

- Link to the specified MULTI instance and host.
- Specify the communication port on the host.
- Set the global timeout to 5 s. The default is 10 s.

Set these properties when you construct the object by entering

```
id = ghsmulti('hostname','localhost','portnum',4444,'timeout',5);
```

The `localhost`, `portnum`, and `timeout` properties are described in Link Properties, as are the other properties for links.

Setting Property Values with set

After you construct an object, the `set` function lets you modify its property values.

Using the `set` function, you can:

- Set link property values
- Display Properties Using set

Example — Setting Link Property Values Using set

To set the timeout specification for the link id from the previous section, enter the following syntax:

```
set(id,'timeout',8);

get(id,'timeout');
ans=

     8
```

The display reflects the changes in the property values.

Retrieving Properties with get

You can use the get command to retrieve the value of an object property.

Example — Retrieving Link Property Values Using get

To retrieve the value of the hostnameproperty for id, and assign it to a variable, enter the following syntax:

```
host=get(id,'hostname')

host =

localhost
```

Direct Property Referencing to Set and Get Values

You can directly set or get property values using MATLAB® structure-like referencing. Do this by using a period to access an object property by name, as shown in the following example.

Example — Direct Property Referencing in Links

To reference an object property value directly, perform the following steps:

1. Create a link with default values.

2. Change its time out and number of open channels.

```
id = ghsmulti;
id.time = 6;
```

Overloaded Functions for ghsmulti Objects

Several methods and functions in Embedded IDE Link™ MU software have the same name as functions in other MathWorks products. These functions behave similarly to their original counterparts, but you apply them to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for objects. After you specify your object by assigning values to its properties, you can apply the methods in this toolbox (such as `address` for reading an address in memory) directly to the variable name you assign to your object. You do not have to specify your object parameters again.

For a list of the methods that act on `ghsmulti` objects, refer to the Chapter 6, "Functions — Alphabetical List" in the function reference pages.

ghsmulti Object Properties

In this section...
"Quick Reference to ghsmulti Properties" on page 2-26
"Details About ghsmulti Object Properties" on page 2-26

Quick Reference to ghsmulti Properties

The following table lists the properties for the links in Embedded IDE Link™ MU. The second column indicates to which object the property belongs. Knowing which property belongs to each object in a link tells you how to access the property.

Property Name	User Settable?	Description
`hostname`	At construction only	Reports the name of the host the Embedded IDE Link™ MU service in Green Hills MULTI® that the object references.
`portnum`	At construction only	Stores the number of the port to communicate with MULTI.
`timeout`	Yes/default	Contains the global timeout setting for the link.

Some properties are read only. Thus, you cannot set the property value. Other properties you can change at any time. If the entry in the User Settable column is "At construction only," you can set the property value only when you create the object. Thereafter, it is read only.

Details About ghsmulti Object Properties

To use the objects for Green Hills MULTI® interface, set values for the following:

- `hostname` — Specify the session with which the object interacts.

- `portnum` — Specify the processor in the session. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — Specify the global timeout value. (Optional. Default is 10 s.)

Details of the properties associated with `ghsmulti` objects appear in the following sections, listed in alphabetical order by property name.

hostname

Property `hostname` identifies the host that is running the Embedded IDE Link™ MU service. Use `hostname` to specify the machine to host your service.

To work with a service, you need the `hostname` and `portnum` values. `Hostname` supports the string `localhost` only.

portnum

Property `portnum` specifies the port for communicating with the Embedded IDE Link™ MU service. MATLAB® uses sockets to communicate withGreen Hills MULTI. The `portnum` property value specifies the port, with a default value of 4444. When you create a new `ghsmulti` object, Embedded IDE Link™ MU assumes the port value is 4444 unless you enter a different value when you configure the software or use the `portnum` input argument with `ghsmulti`.

timeout

Property `timeout` specifies how long Green Hills MULTI® waits for any process to finish. You set the global timeout when you create an object for a session in Green Hills MULTI®. The default global timeout value 10 s. The following example shows the `timeout` value for object `id2`.

```
display(id2)

MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\ppc\
```

Project Generator

Introducing Project Generator (p. 3-2)	Describes code generation with Project Generator.
Using the Embedded IDE Link™ MU Blockset (p. 3-3)	Describes the contents of the multilinklib blockset.
Schedulers and Timing (p. 3-9)	Describes the timer-based and asynchronous schedulers.
Project Generator Tutorial (p. 3-18)	Takes you through the process of creating models in Simulink® and generating code for your processors.
Setting Real-Time Workshop® Code Generation Options for Supported Processors (p. 3-27)	Provides details about setting Real-Time Workshop® software options when you generate code from Simulink® models to supported processors.
Setting Real-Time Workshop® Category Options (p. 3-30)	Information about setting code generation options for models.
Model Reference and Embedded IDE Link™ MU Software (p. 3-45)	Introduces model reference and how you use model reference with Embedded IDE Link™ MU software.

Introducing Project Generator

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

- Automated project building for Green Hills MULTI® that lets you create MULTI projects from code generated by Real-Time Workshop® and Real-Time Workshop Embedded Coder™. Project generator populates projects in the MULTI development environment.
- Blocks in the library `multilinklib` for controlling the scheduling and timing in generated code.
- Highly configurable code generation using model configuration parameters and target preferences block options.
- Ability to use Embedded IDE Link™ MU with one of two system target files to generate code specific to your processor.
- Highly configurable project build process.
- Automatic downloading and running of your generated projects on your processor.

To configure your Simulink® models to use the Project Generator component, do one or both of the following tasks:

- Add a Target Preferences block from the Embedded IDE Link™ MU blockset (`multilinklib`) to the model.
- To use the asynchronous scheduler capability in Embedded IDE Link™ MU, add a hardware interrupt block or idle task block from the blockset `multilinklib`.

The following sections describe the blockset and the blocks in it, the scheduler, and the Project Generator component.

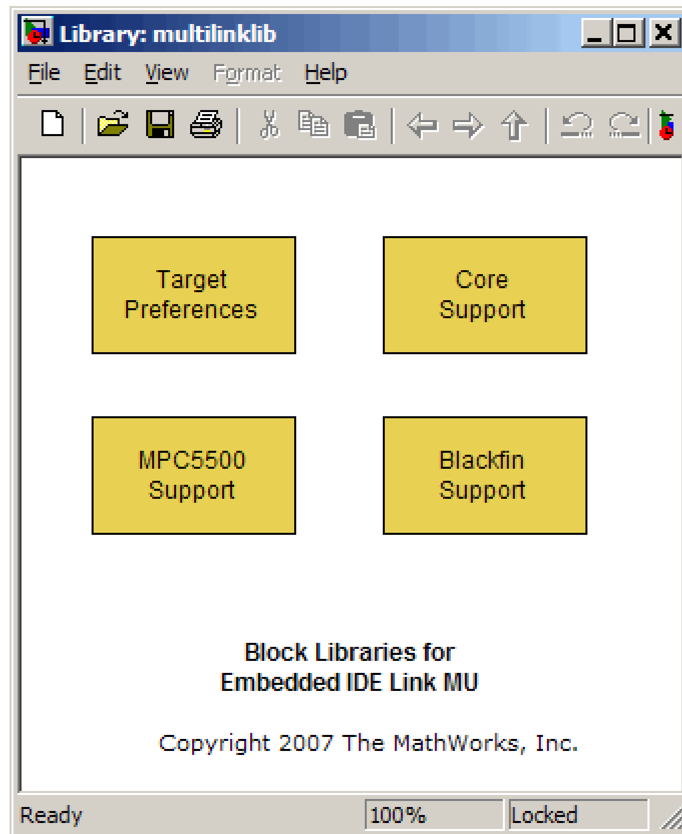
Using the Embedded IDE Link™ MU Blockset

Embedded IDE Link™ MU block library `multilinklib` comprises block libraries that contain blocks designed for generating projects for specific processors. The following table describes these libraries.

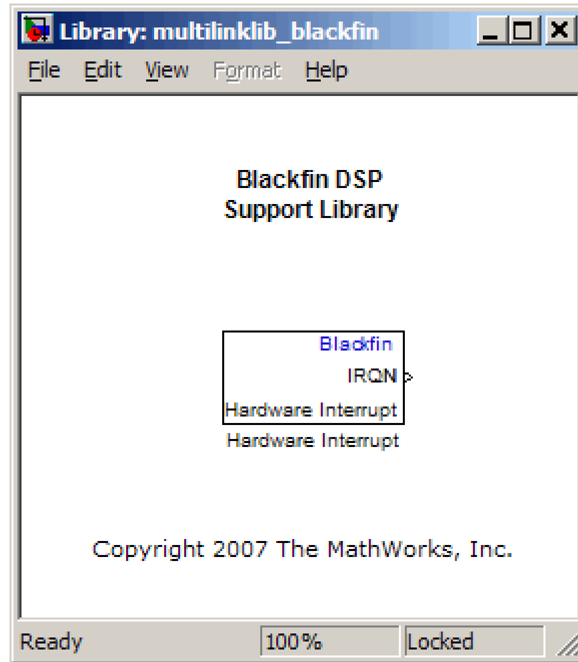
Library	Description
Blackfin DSP Support (`multilinklib_blackfin`)	Block for task scheduling on Analog Devices® Blackfin® processors
Core Support (`multilinklib_coresupport`)	Blocks for task scheduling and manipulating memory on supported processors
Target Preferences (`multilinklib_tgtprefs`)	Block that configures models for supported processors
MPC5500 Support (`multilinklib_mpc5500`)	Block for task scheduling on Freescale™ MPC5500 processors

Blocks for the processor families are almost identical. Each block has a reference page that describes the options for the block. Use the Help browser to get more information about a block shown in any of the following figures.

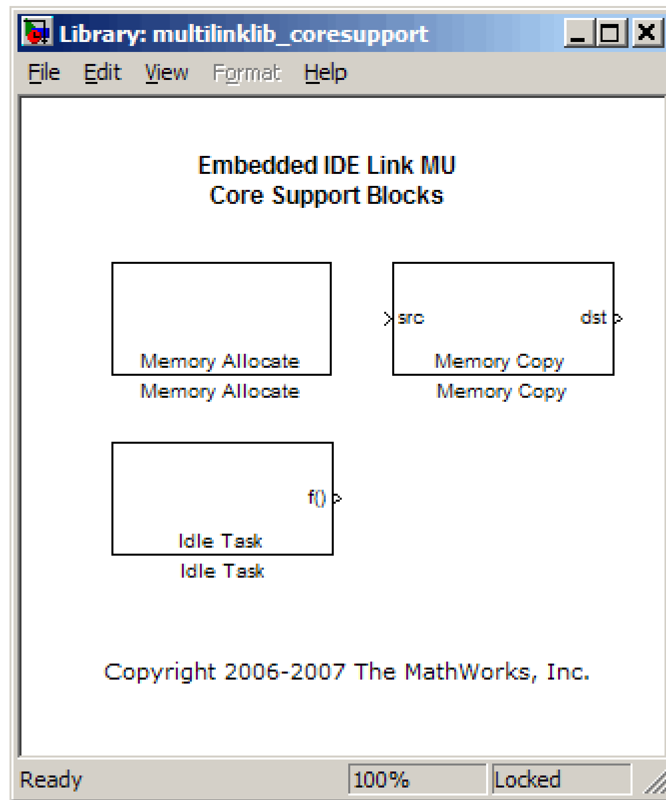
The first figure shows the main library of libraries in Embedded IDE Link™ MU.


Library: multilinklib
File Edit View Format Help
Target Preferences
Core Support
MPC5500 Support
Blackfin Support
Block Libraries for
Embedded IDE Link MU
Copyright 2007 The MathWorks, Inc.
Ready
100%
Locked

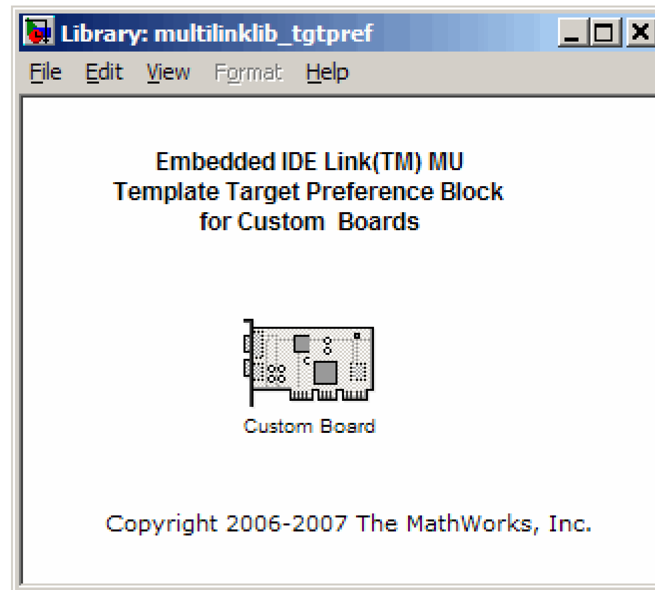
The next figure shows the Blackfin Support library.



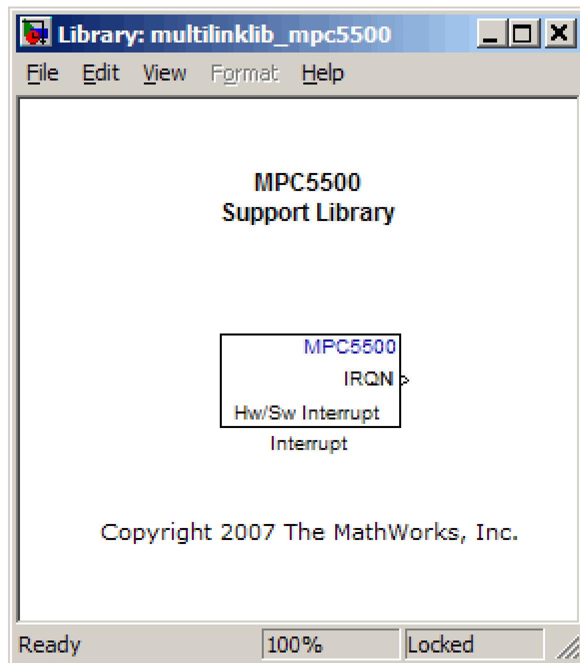
The Core Support library contains the blocks shown in the next figure.



The target preferences library for all supported processors appears in the next figure.



The MPC5500 Support library appears in the next figure.


Library: multilinklib_mpc5500
File
Edit
View
Format
Help
MPC5500
Support Library
MPC5500
IRQN
Hw/Sw Interrupt
Interrupt
Copyright 2007 The MathWorks, Inc.
Ready
100%
Locked

Schedulers and Timing

In this section...

"Timer-Based Versus Asynchronous Interrupt Processing" on page 3-9

"Synchronous Scheduling" on page 3-10

"Asynchronous Scheduling" on page 3-11

"Scheduling Blocks" on page 3-11

"Asynchronous Scheduler Examples" on page 3-12

"Uses for Asynchronous Scheduling" on page 3-15

The following sections describe how Embedded IDE Link™ MU provides timing and scheduling for generated code running on your processor.

Timer-Based Versus Asynchronous Interrupt Processing

Code generated for periodic tasks, both single- and multitasking, runs out of the context of a timer interrupt. The code generated by model blocks for periodic tasks runs periodically. A periodic interrupt with period equal to the model base sample time clocks the generated code.

Note In timer-based models, the timer counts through one full base-sample-time before it creates an interrupt. When the model is finally executed, it is for time 0.

This execution scheduling scheme is not flexible enough for some systems, such as control and communication systems that must respond to asynchronous events in real time. Such systems often handle a variety of hardware interrupts in an asynchronous, or aperiodic, fashion.

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

If your application processes hardware interrupts asynchronously, add the appropriate asynchronous scheduling blocks from the Embedded IDE Link™ MU library to your model. The following table lists the scheduling blocks.

Processor Support Library	Block Purpose	Description
Blackfin	Hardware Interrupt for asynchronous scheduling	Create interrupt service routine to handle hardware interrupt on Analog Devices® Blackfin® processors
Core DSP	Idle Task	Create task that runs as separate thread for any Green Hills Software supported processor
Target Preferences	Target Preferences	Configure model for Green Hills Software supported processor
MPC55xx	Hardware Interrupt for asynchronous scheduling	Create interrupt service routine to handle hardware interrupt on Freescale™ processors

If your application does not service asynchronous interrupts, include in your model only the algorithm and device driver blocks that specify the periodic sample times. Generating code from a model configured this way automatically enables and manages a timer interrupt. The periodic timer interrupt clocks the entire model.

Synchronous Scheduling

For code that runs synchronously in the context of the timer interrupt, each model iteration runs after an interrupt service routine (ISR) services a posted interrupt. The code generated for Embedded IDE Link™ MU uses Timer 1. Timer 1 is configured so that the base rate sample time for the coded process corresponds to the interrupt rate. Embedded IDE Link™ MU calculates and configures the timer period to ensure the correct sample rate.

The minimum achievable base rate sample time depends on the algorithm complexity and the CPU clock speed. The maximum value depends on the maximum timer period value and the CPU clock speed.

Simulink® assigns a default sample time of 0.2s in the following case:

- All the blocks in the model inherit their sample time
- The sample time is not defined explicitly in the model

Asynchronous Scheduling

Embedded IDE Link™ MU provides the interrupt and task scheduling blocks shown in the following table to help you model and automatically generate code for asynchronous systems.

Mode	Block
Hardware or Software (MPC5500 processors only) Interrupt	Blackfin, MPC5500
Free-Running Task for Bare-Board Code Generation	Idle Task

- Any hardware interrupt block, such as
  - Hardware Interrupt
  - HW/SW Interrupt

  and a block for bare-board code generation mode
- Idle Task

Scheduling Blocks

Embedded IDE Link™ MU Hw/Sw Interrupt blocks perform the following functions:

- Enable selected hardware interrupts for the supported processors
- Generate corresponding ISRs
- Connect the ISRs to the corresponding interrupt service vector table entries

When you connect the output of the Hw/Sw Interrupt block to the control input of a function-call subsystem, the ISRs call the generated subsystem code each time the enabled interrupt is raised.

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

Asynchronous Scheduler Examples

After you identify the blocks to use, you can use an asynchronous (real-time) scheduler for your application. With the asynchronous scheduler, you can schedule the execution of interrupts and tasks by using blocks from the following libraries:

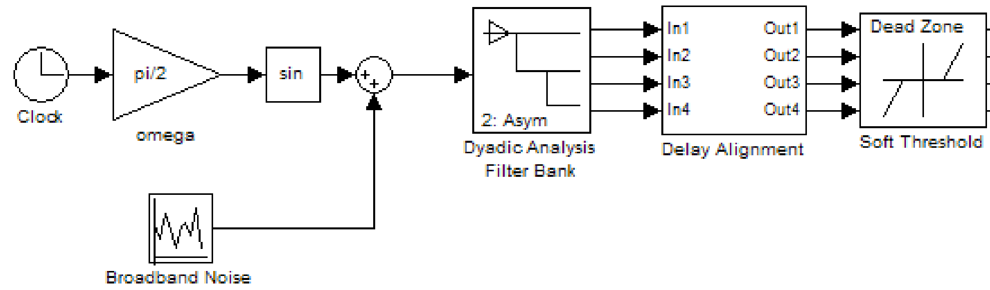
- Core Support
- Analog Devices® Blackfin® Support
- Freescale™MPC5500 Support

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

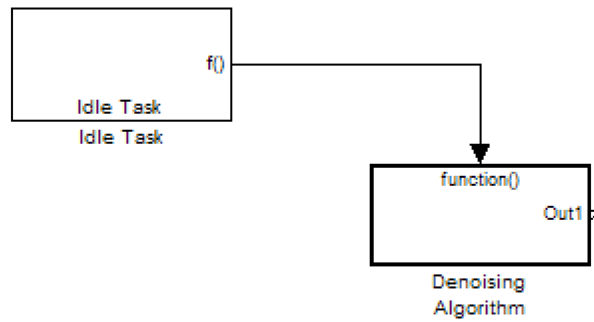
The following figures show a model updated to use the asynchronous scheduler rather than the synchronous scheduler.

Note You cannot build or run the example models without additional blocks. They provide example configurations only.

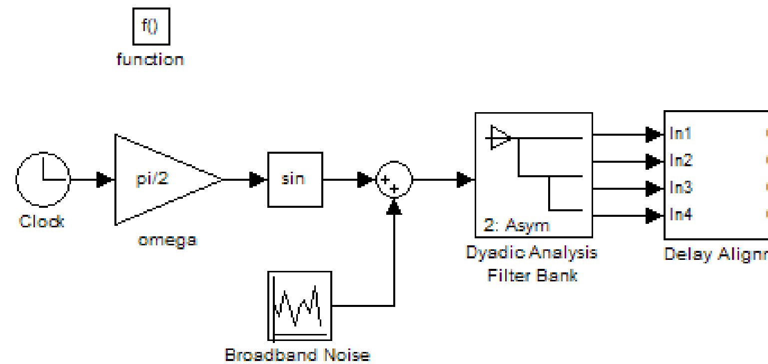
Before



After



Model Inside the Function Call Subsystem Block

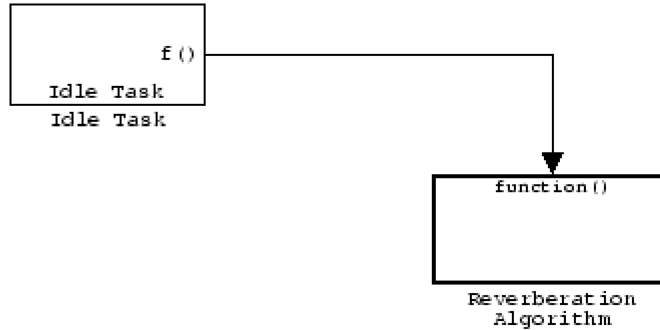


Uses for Asynchronous Scheduling

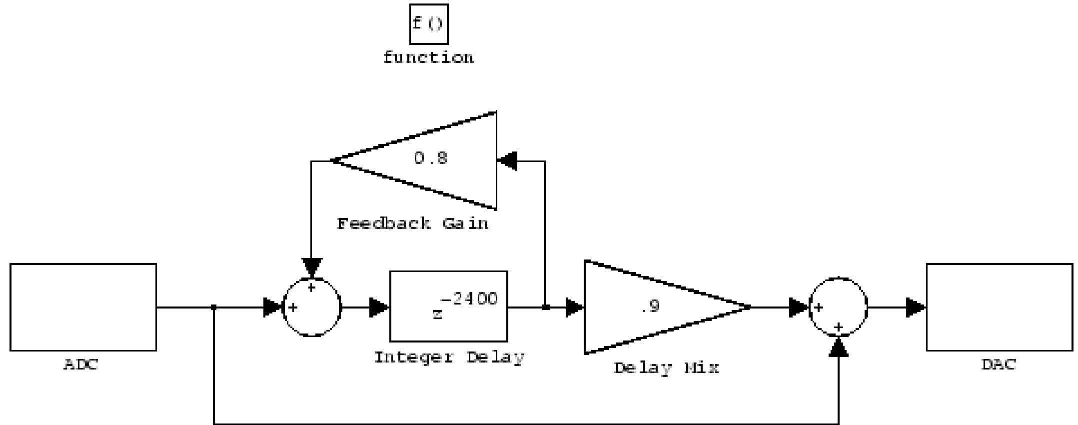
The following sections show common cases for the scheduling blocks described in the previous sections.

Idle Task

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

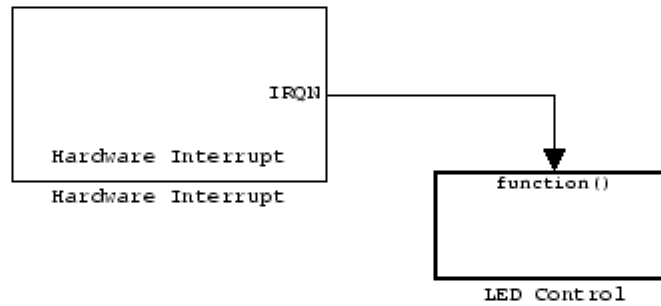


The function generated for this task normally runs in free-running mode—repetitively and indefinitely. However, the input and output blocks in this subsystem run in blocking mode. As a result, subsystem execution of the reverberation function is the same as the subsystem described for the Free-Running Task. Task execution is data driven via a background DMA interrupt-controlled ISR, shown in the following figure.



Hardware Interrupt Triggered Task

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



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

Project Generator Tutorial

In this section...

"Process for Building and Generating a Project" on page 3-18

"Create the Model" on page 3-19

"Adding the Target Preferences Block to Your Model" on page 3-20

"Specifying Simulink® Configuration Parameters for Your Model" on page 3-23

"Creating Your Project" on page 3-25

Process for Building and Generating a Project

In this tutorial, you build a model and generate a project from the model into Green Hills MULTI®.

Note The model shows project generation only. You cannot build and run the model on your processor without additional blocks.

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

1. Use Simulink® blocks, Signal Processing Blockset blocks, and blocks from other blocksets to create the model application.

2. Add the target preferences block from the Embedded IDE Link™ MU Target Preferences library to your model.

3. Double-click the Target Preferences block to open the block dialog box.

4. Select your processor from the **Processor** list. Verify and set the block parameters for your hardware, such as **CPU clock** and the options on the **Memory** and **Section** panes. In most cases, the default settings for the selected processor work fine.

5. Set the configuration parameters for your model, including the following parameters:

- Solver parameters such as simulation start and solver options. Choose the discrete solver.
- Real-Time Workshop® options such as processor configuration and processor compiler selection

6 Generate your project.

7 Review your project in MULTI.

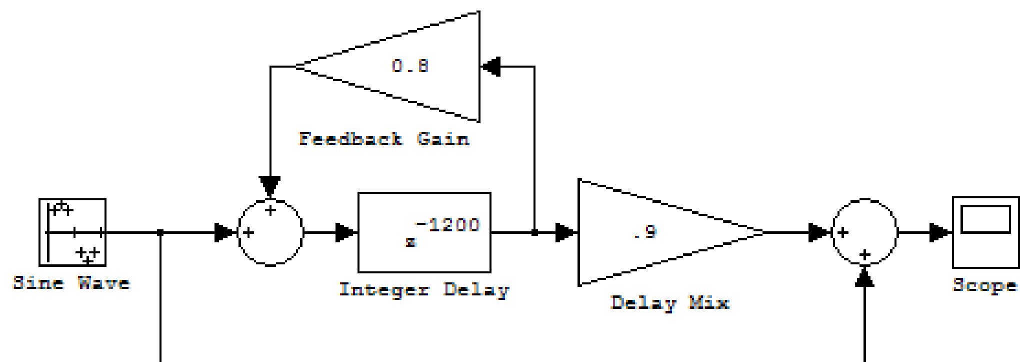
Create the Model

To build the model for this tutorial, follow these steps:

1 Start Simulink®.

2 Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.

3 Use Simulink® blocks and Signal Processing Blockset blocks to create the following model.



Look for the Integer Delay block in the Discrete library of Simulink® and the Gain block in the Commonly Used Blocks library. This model implements an audio signal reverberation scheme. Part of the input audio signal passes directly to the output. A delayed version passes through

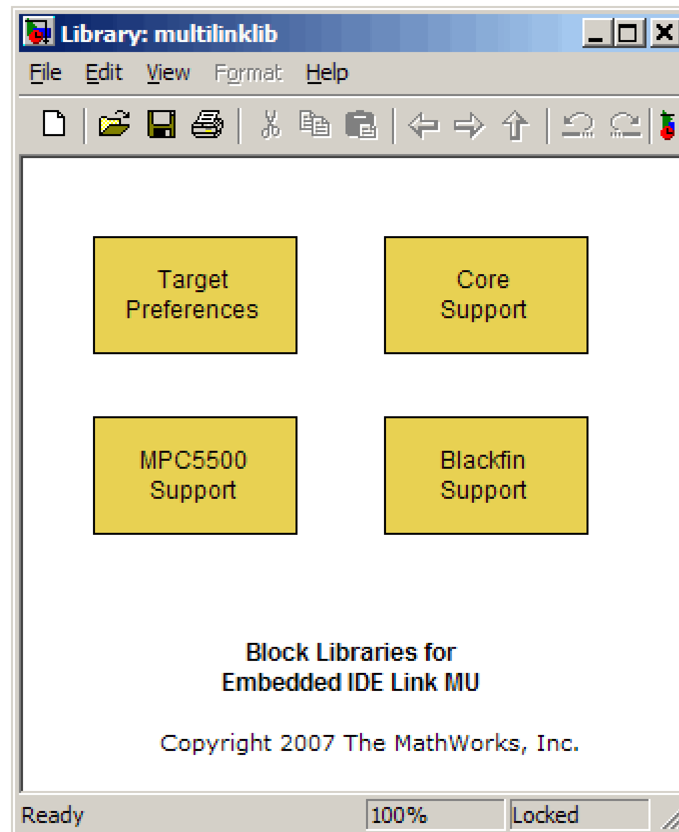
a feedback loop before reaching the output. The result is an echo, or reverberation, added to the audio output.

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

Adding the Target Preferences Block to Your Model

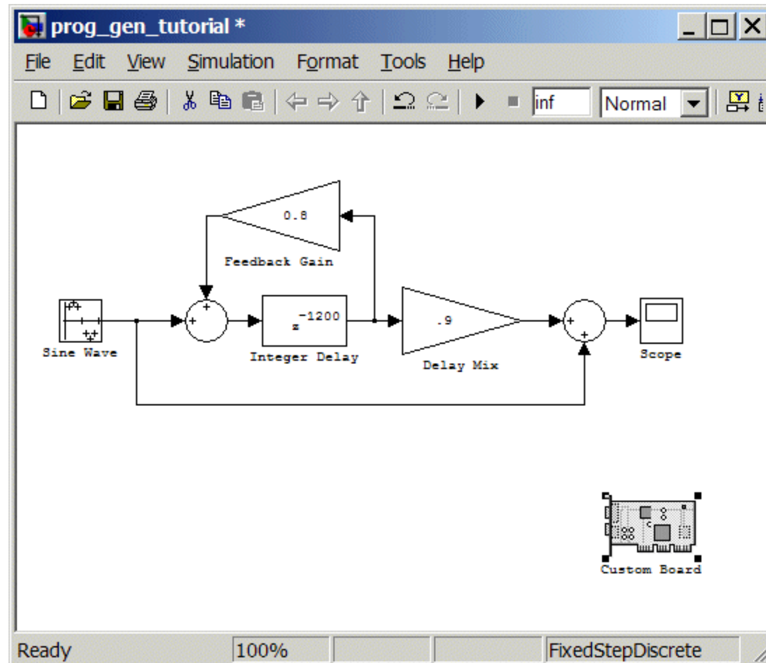
So that you can configure your model to work with the supported processors, the software includes the Target Preferences block library. The library contains the Custom Board block that you use to configure models for any of the supported processors.

Entering `multilinklib_tgtpref` at the MATLAB® prompt opens this window showing the block library. This block library is included in the Embedded IDE Link™ MU `multilinklib` blockset in the Simulink® Library browser.



To add the Target Preferences block to your model, follow these steps:

1. Double-click Embedded IDE Link™ MU in the Simulink® Library browser to open the `multilinklib` blockset.
2. Double-click the library Target Preferences to see the Custom Board block.
3. Drag and drop the Custom Board block to your model as shown in the following figure.



4 Double-click the Custom Board block to open the block dialog box.

5 In the Block dialog box, select your processor from the **Processor** list.

6 Check the **CPU clock** value and change it if necessary to match your processor clock rate.

7 Review the settings on the **Memory** and **Sections** tabs to verify that they are correct for the processor you selected.

8 Click **OK** to close the Target Preferences dialog box.

You have completed the model. Next, configure the model configuration parameters to generate a project in Green Hills MULTI® from your model.

Specifying Simulink® Configuration Parameters for Your Model

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

Setting Solver Options

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

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

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

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

Setting Real-Time Workshop® Code Generation Options

To configure Real-Time Workshop® software to use the correct processor files, compile your model, and run your model executable file, set the options in the

Real-Time Workshop category of the model Configuration Parameters. Follow these steps to set the Real-Time Workshop® software options to generate code tailored for your processor:

1. Select Real-Time Workshop on the **Select** tree.
2. In **Target selection**, click **Browse** to select the appropriate system target file for code generation—multilink_grt.tlc or multilink_ert.tlc (if you use Real Time Workshop Embedded Coder software). The correct target file might already be selected.

   Clicking **Browse** opens the **System Target File Browser** to allow you to change the system target file.
3. On the **System Target File Browser**, select the proper system target file multilink_grt.tlc or multilink_ert.tlc, and click **OK** to close the browser.

Setting Embedded IDE Link™ MU Code Generation Options

After you set the Real-Time Workshop® options for code generation, set the options that apply to your Embedded IDE Link™ MU software run-time and build processes.

1. From the **Select** tree, choose Embedded IDE Link MUto specify code generation options that apply to the processor.
2. Set the following **Runtime** options:
   - **Build action**: Create_project.
   - **Interrupt overrun notification method**: Print_message.
3. (optional) Under **Link Automation**, verify that **Export MULTI link handle to base workspace** is selected and provide a name for the handle in **MULTI link handle name**.
4. Set the following options in the dialog box under **Project options**:
   - Set **Compiler options string** to blank.
5. Under **Code Generation**, select the **Inline run-time library functions** option. Clear all other options.

6 Change the category on the **Select** tree to `Hardware Implementation`.

7 Verify that the Device type is the correct value for your processor—`Analog Devices`, `NEC`, or `Freescale`.

You have configured the Real-Time Workshop® options that let you generate a project for your processor. A few Real-Time Workshop® categories on the **Select** tree, such as `Comments`, `Symbols`, and `Optimization` do not require configuration for use with Embedded IDE Link™ MU software. In some cases, set options in the other categories to configure other code generation features.

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

Creating Your Project

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

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

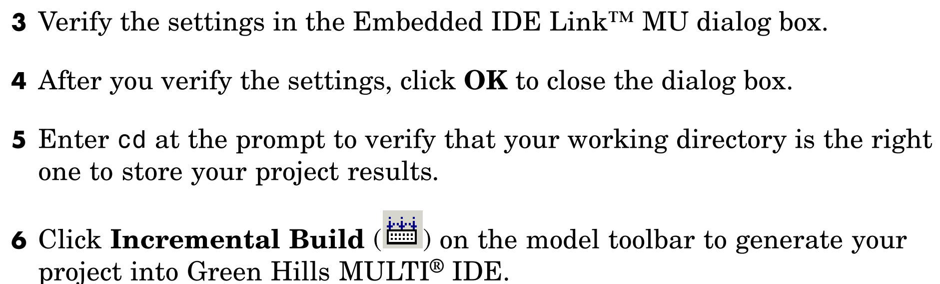
2 To verify that you configured your Embedded IDE Link™ MU software correctly, issue the following command at the prompt to open the Embedded IDE Link™ MU Configuration dialog box.

```
ghsmulticonfig
```

3 Verify the settings in the Embedded IDE Link™ MU dialog box.

4 After you verify the settings, click **OK** to close the dialog box.

5 Enter `cd` at the prompt to verify that your working directory is the right one to store your project results.

6 Click **Incremental Build** () on the model toolbar to generate your project into Green Hills MULTI® IDE.

When you press with `Create_project` selected for **Build action**, the build process starts the Green Hills MULTI® application and populates a new project.

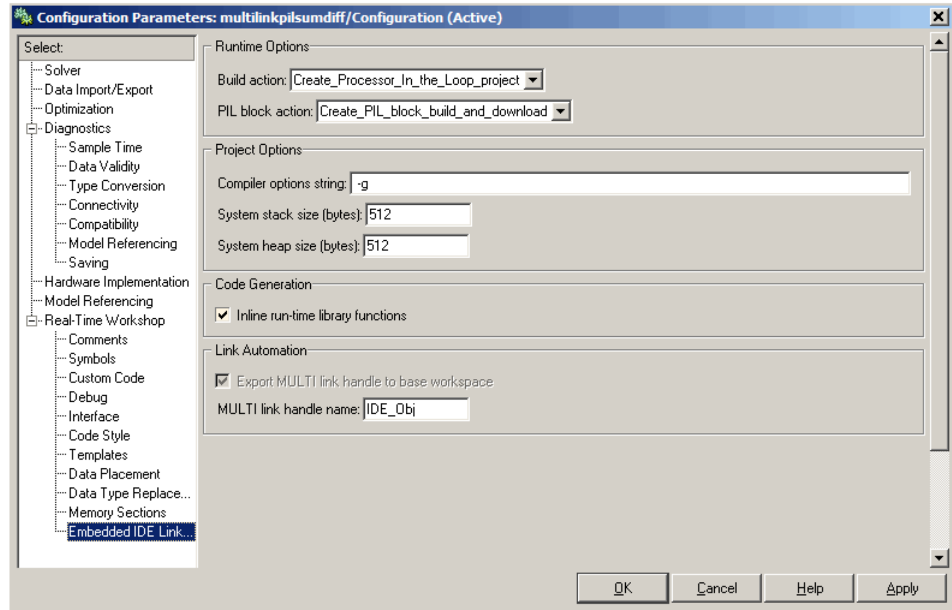
Setting Real-Time Workshop® Code Generation Options for Supported Processors

If the model contains continuous-time states, set the fixed-step solver step size and specify an appropriate fixed-step solver before you generate code. At this time, select an appropriate sample rate for your system. Refer to the *Real-Time Workshop® User's Guide* for additional information.

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

To open the Configuration Parameters dialog box for your model, select **Simulation > Configuration Parameters** from the menu bar.

The following figure shows the Real-Time Workshop® **Select** tree categories when you are using Embedded IDE Link™ MU.



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

One category under `Real-Time Workshop` is specific to Embedded IDE Link™ MU and appears when you select either the `multilink_grt.tlc` or `multilink_ert.tlc` system target file.

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

For Embedded IDE Link™ MU software, the target file to select is `multilink_grt.tlc`. Selecting either the `multilink_grt.tlc` or `multilink_ert.tlc` adds categories to the **Select** tree that are specific to generating code with Embedded IDE Link™ MU software. The `multilink_grt.tlc` file is appropriate for all projects.

Select `multilink_ert.tlc` when you are developing projects or code for embedded processors (requires Real-Time Workshop® Embedded Coder™

software) or you plan to use Processor-in-the-Loop features (requires Real-Time Workshop® Embedded Coder™ software).

The following sections describe each Real-Time Workshop® category and the options available in each.

Setting Real-Time Workshop® Category Options

In this section...

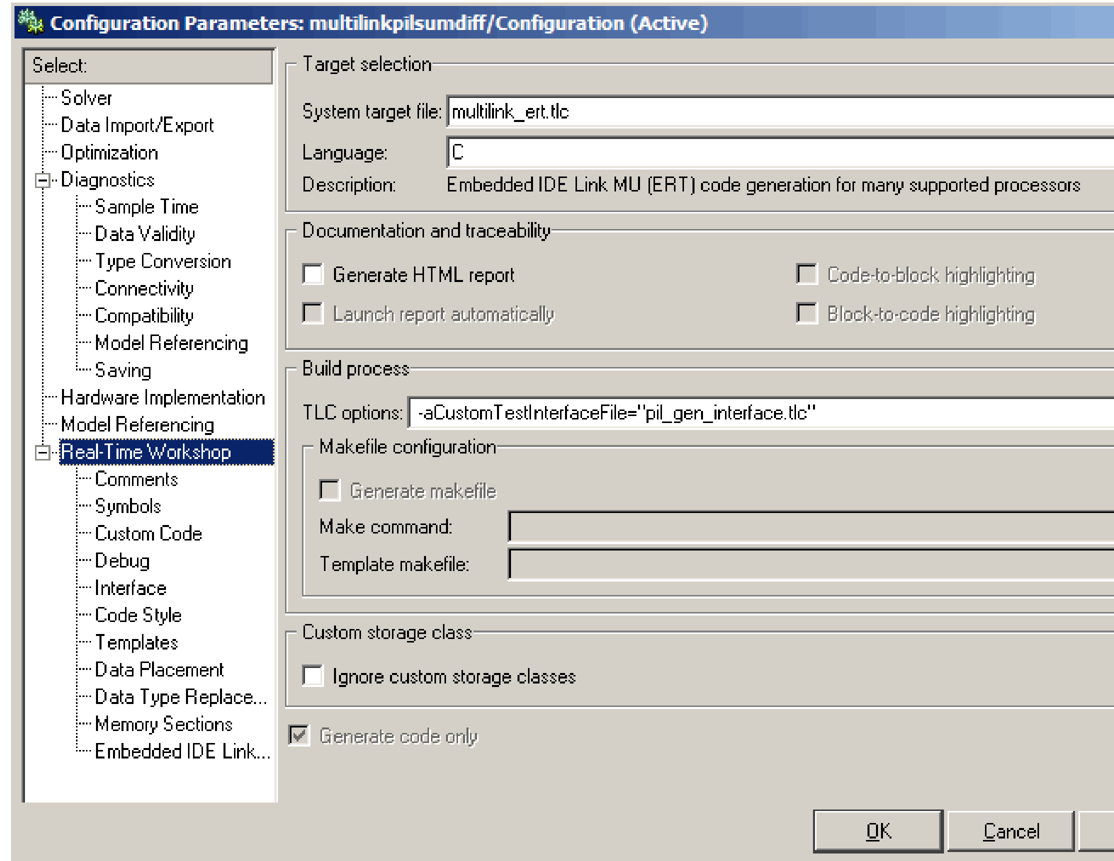
"About Select Tree Category Options" on page 3-30
"Target Selection" on page 3-31
"Documentation" on page 3-32
"Build Process" on page 3-33
"Custom Storage Class" on page 3-33
"Debug Pane Options" on page 3-34
"Optimization Pane Options" on page 3-35
"Embedded IDE Link™ MU Pane Options" on page 3-37
"Overrun Indicator and Software-Based Timer" on page 3-44

About Select Tree Category Options

Use the options in the **Select** tree under `Real-Time Workshop` to perform the following configuration tasks:

- Specify your processor
- Specify your documentation needs.
- Configure your build process.
- Specify whether to use custom storage classes.

When you select one of the Embedded IDE Link™ MU system target files, the Embedded IDE Link™ MU category appears in the **Select** tree as shown in the following figure.



Target Selection

The following parameter enables you to select your system target file to support code generation with Embedded IDE Link™ MU software.

System target file

Clicking **Browse** opens the Target File Browser where you select `multilink_grt.tlc` as your `Real-Time Workshop` **System target file** for Embedded IDE Link™ MU. When you select the Embedded IDE Link™ MU target file, Real-Time Workshop® disables the makefile configuration options.

Embedded IDE Link™ MU software does not use makefiles. Embedded IDE Link™ MU creates and uses MULTI projects directly.

If you are using Real-Time Workshop® Embedded Coder™ software, select the `multilink_ert.tlc` target file in **System target file**.

Documentation

Two options control HTML report generation during code generation.

- "Generate HTML report" on page 3-32
- "Launch report automatically" on page 3-32

Generate HTML report

After you generate code, this option tells the software whether to generate an HTML report that documents the C code generated from your model. When you select this option, Real-Time Workshop® writes the code generation report files in the `html` subdirectory of the build directory. The top-level HTML report file is named *`modelname`*`_codegen_rpt.html` or *`subsystemname`*`_codegen_rpt.html`. For more information about the report, refer to the online help for Real-Time Workshop®. You can also use the following command at the MATLAB® prompt to get more information.

```
docsearch 'Generate HTML report'
```

When you select **Block-to-code highlighting** and **Code-to-block highlighting**, your HTML report adds hyperlinks to various features in your Simulink® model.

Launch report automatically

This option directs Real-Time Workshop® to open a MATLAB® Web browser window and display the code generation report. If you clear this option, you can open the code generation report (*`modelname`*`_codegen_rpt.html` or *`subsystemname`*`_codegen_rpt.html`) manually in a MATLAB® Web browser window or in another Web browser.

Build Process

Parameters in this group determine how Real-Time Workshop® software builds the makefile for the generated code.

- “Template makefile” on page 3-33
- “Make command” on page 3-33

Template makefile

Not used.

Make command

Not used.

Custom Storage Class

Use the parameter in this group to specify whether to use custom storage classes. For more information about custom storage classes, refer to the Real-Time Workshop® documentation.

Ignore custom storage classes

When you generate code from a model that uses custom storage classes (CSC), clear **Ignore custom storage classes**. This setting is the default value for Embedded IDE Link™ MU software and for Real-Time Workshop® Embedded Coder™ software.

When you select **Ignore custom storage classes**, storage class attributes and signals are affected in the following ways:

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

Ignore custom storage classes lets you switch to a processor that does not support CSCs, such as the generic real-time target (GRT), without reconfiguring your parameter and signal objects.

Generate code only

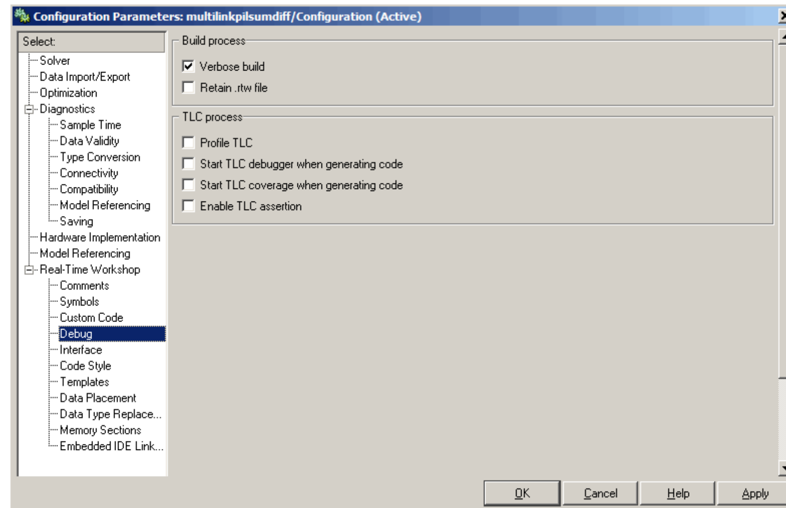
The **Generate code only** option does not apply to targeting with Embedded IDE Link™ MU. To generate source code without building and executing the code on your processor, select `Embedded IDE Link MU` from the **Select** tree. Then, under **Runtime**, select `Create_project` for **Build action**.

Debug Pane Options

Real-Time Workshop® uses the Target Language Compiler (TLC) to generate C code from the *`model`*`.rtw` file. The TLC debugger helps you identify programming errors in your TLC code. Using the debugger, you can perform the following actions:

- View the TLC call stack.
- Execute TLC code line-by-line.
- Analyze or change variables in a specified block scope.

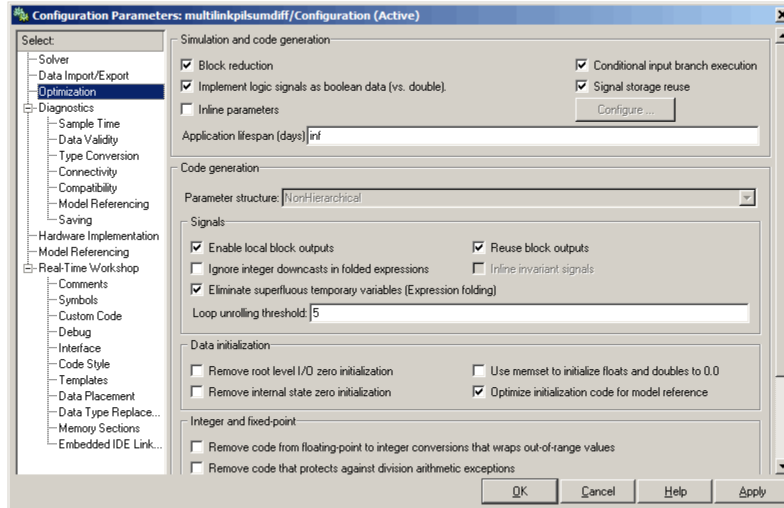
When you select `Debug` from the **Select** tree, you see the `Debug` options as shown in the next figure. In this dialog box, you set options that are specific to Real-Time Workshop® process and TLC debugging.



For details about using the options in `Debug`, refer to “About the TLC Debugger” in your Real-Time Workshop® Target Language Compiler documentation.

Optimization Pane Options

On the `Optimization` pane in the Configuration Parameters dialog box, you set options for the code that Real-Time Workshop® generates during the build process. Use these options to tailor the generated code to your needs. Select `Optimization` from the **Select** tree on the Configuration Parameters dialog box. The figure shows the `Optimization` pane when you select the system target file `multilink_grt.tlc` under **Real-Time Workshop system target file**.



These options are typically selected for Real-Time Workshop® software to provide optimized code generation for common code operations:

Parameter	Description
Conditional input branch execution	Improve model execution when the model contains Switch and Multiport Switch blocks.
Signal storage reuse	Reuse signal memory.
Enable local block outputs	Specify whether block signals are declared locally
Reuse block outputs	Specify whether Real-Time Workshop® reuses signal memory.
Eliminate superfluous temporary variables (Expression folding)	Collapse block computations into single expressions.

Parameter	Description
Loop unrolling threshold	Specify the minimum signal or parameter width that generates a `for` loop.
Optimize initialization code for model reference	Specify whether to generate initialization code for blocks that have states.

For more information about using these and the other `Optimization` options, refer to the Real-Time Workshop® documentation.

Embedded IDE Link™ MU Pane Options

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

Parameter	Description
Runtime Options	Set options for run-time operations, like the build action and whether to use processor-in-the-loop functionality.
Project Options	Set the build options for your project code generation, including compiler and linker settings.
Code Generation	Configure your code generation needs, such as enabling real-time task execution profiling.
Link Automation	Specify whether to export the `ghsmulti` object to the MATLAB® workspace.

Runtime Options

Before you run your model as an executable on any Green Hills Software processor, configure the run-time options for the model.

By selecting values for the options available, you configure the model build process and task or process overrun handling.

Build action

To specify to Real-Time Workshop® what to do when you click **Build**, select one of the following options. The actions are cumulative—each listed action adds features to the previous action on the list and includes all the previous features:

Build Action Selection	Description
`Create_project`	Directs Real-Time Workshop® software to start Green Hills MULTI® software and populate a new project with the files from the build process. This option offers a convenient way to build projects in Green Hills MULTI® IDE.Real-Time Workshop® software generates C code only from the model. It does not use the Green Hills Software development tools, such as the compiler and linker. Also, MATLAB® software does not create the `ghsmulti` object for accessing the Green Hills MULTI® software that results from the other options.
`Archive_library`	Directs Real-Time Workshop® software to archive the project for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your Green Hills MULTI® projects for models that you use in model referencing.

Build Action Selection	Description
Build	Builds the processor-specific executable file, but does not download the file to your processor.
Create_processor_in_the_loop_project	Directs the Real-Time Workshop® software code generation process to create PIL algorithm object code as part of the project build.
Build_and_execute	Directs Real-Time Workshop® software to build, download, and run your generated code as an executable on your processor.

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

To run your model on the processor, select the default build action, `Build_and_execute`. Real-Time Workshop® then automatically downloads and runs the model on your processor.

Note When you build and execute a model on your processor, the Real-Time Workshop® software build process resets the processor automatically.

Interrupt overrun notification method

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

`None`	Ignore overruns encountered while running the model.
`Print_message`	When the processor encounters an overrun condition, it prints a message to the standard output device, `stdout`.
`Call_custom_function`	Respond to overrun conditions by calling the custom function you identify in **Interrupt overrun notification function**.

Interrupt overrun notification function

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

PIL block action

Selecting `Create_Processor_In_the_Loop_project` for the **Build action** enables **PIL block action**. Choose one of the following three actions for creating a PIL block:

PIL Block Action Selection	Description
`None`	Do not create the PIL block or PIL algorithm object code.
`Create PIL block`	Create the algorithm object code and PIL block. Use this selection to create a PIL block.
`Create PIL block_build_and_download`	Create the algorithm object code and PIL block, and then build and download the project to your processor. Use this selection to update an existing PIL block in a model.

Project Options

Before you run your model as an executable on any processor, configure the Project options for the model. By default, the setting for the project options is `Custom`, which applies MathWorks specified compiler and linker settings for your generated code.

Compiler options string

To determine the degree of optimization provided by the Green Hills® optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your Green Hills® MULTI® documentation. When you create new projects, Embedded IDE Link™ MU software sets the optimization to `-g`.

System stack size (bytes)

Enter the amount of memory to use for the stack. For more information on memory needs, refer to **Enable local block outputs** on the **Optimization** pane of the dialog box. The block output buffers are placed on the stack until the stack memory is fully allocated. When the stack memory is full, the output buffers go in global memory. Refer to the online Help system for more information about Real-Time Workshop® options for configuring and building models and generating code.

Code Generation

From this category, you select options that define the way your code is generated:

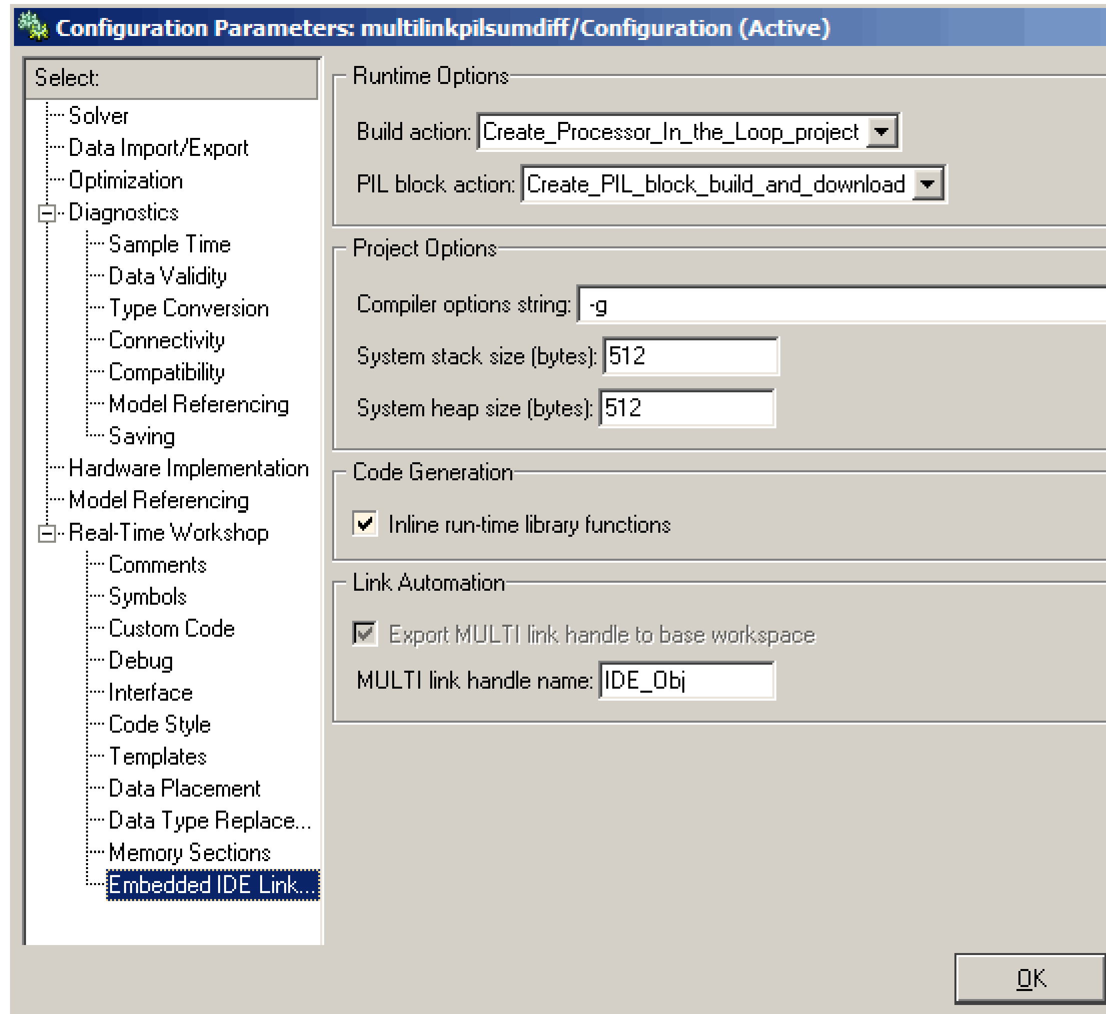
Parameter	Description
Profile real-time task execution	Enable real-time task execution profiling in your project.
Inline run-time library functions	Specify whether to inline each Signal Processing blockset and Video and Imaging blockset function.

To enable the real-time execution profile capability, select **Profile real-time task execution**. When you select this option, the build process instruments your code to provide performance profiling at the task level. When you run

your code, the executed code reports the profiling information in graphical presentation and an HTML report forms.

To specify whether the functions generated from blocks in your model are used inline or by pointers, select **Inline run-time library functions**. Selecting this option tells the compiler to inline each Signal Processing blockset and Video and Imaging blockset function. Using inline functions optimizes your code to run more efficiently. However, such optimization requires more memory.

As shown in the following figure, the default setting uses inlining to optimize your generated code.



When you designate a block function as inline, the compiler replaces each call to a block function with the equivalent function code from the static run-time library. If your model uses the same block four times, your generated code contains four copies of the function.

While this redundancy uses more memory, inline functions run more quickly than calls to the functions outside the generated code.

Link Automation

When you use Real-Time Workshop® software to build a model to a processor, Embedded IDE Link™ MU software makes a connection between MATLAB® and Green Hills® MULTI®. MATLAB® represents that connection as a `ghsmulti` object. The properties of the `ghsmulti` object contain information about the IDE instance it refers to, such as the session and processor it accesses. In this pane, the **Export MULTI link handle to base workspace** option instructs the software to export the `ghsmulti` object created during code generation to your MATLAB® workspace. MATLAB® exports the object with the name you specify in **MULTI link handle name**.

Overrun Indicator and Software-Based Timer

Embedded IDE Link™ MU software includes software that generates interrupts in models that use multiple clock rates. In the following cases, the overrun indicator does not work:

- In multirate systems where the rate in the model is not the same as the base clock rate for your model. In such cases, the timer in Embedded IDE Link™ MU provides the interrupts for setting the model rate.
- In models that do not include ADC or DAC blocks. In such cases, the timer provides the software interrupts that drive model processing.

Model Reference and Embedded IDE Link™ MU Software

In this section...

"About Model Reference" on page 3-45

"How Model Reference Works" on page 3-45

"Using Model Reference with Embedded IDE Link™ MU Software" on page 3-47

"Configuring Targets to Use Model Reference" on page 3-48

About Model Reference

Model reference lets your model include other models as modular components. This technique is useful because it provides the following capabilities:

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

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

How Model Reference Works

Model reference behaves differently in simulation and in code generation. This discussion uses the following terms:

- The *Top model* is the root model block or model. It refers to other blocks or models. In the model hierarchy, this model is the topmost model.

- *Referenced models* are blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

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

Model Reference in Simulation

When you simulate the top model, Real-Time Workshop® detects that your model contains referenced models. Simulink® generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (`.mex` file) for each reference model that is used to simulate the top model.

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

Model Reference in Code Generation

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

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

After building all the referenced models, Real-Time Workshop® calls `make_rtw` on the top model. The call to `make_rtw` links to the library files Real-Time Workshop® created for the associated referenced models.

Using Model Reference with Embedded IDE Link™ MU Software

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

Build Action Setting

The most important requirement for using model reference with the Green Hills® MULTI® software supported processors is you must set the **Build action** (select **Configuration Parameters > Embedded IDE Link MU**) for all models referred to in the simulation to `Archive_library`.

To set the build action, perform the following steps:

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

   The Configuration Parameters dialog box opens.
3. From the **Select** tree, choose `Embedded IDE Link MU`.
4. In the right pane, under **Runtime**, select set `Archive_library` from the **Build action** list.

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

Selecting `Archive_library` disables the **Interrupt overrun notification method**, **Export MULTI link handle to the base workspace**, and **System stack size** options for the referenced models.

Target Preferences Blocks in Reference Models

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

The referenced models need target preferences blocks to provide information about which compiler and which archiver to use. Without these blocks, the compile and archive processes do not work.

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

Other Block Limitations

Model reference with Embedded IDE Link™ MU software code generation options does not allow you to use noninlined S-functions in reference models. Verify that the blocks in your model do not use noninlined S-functions.

Configuring Targets to Use Model Reference

When you create models to use in Model Referencing, keep in mind the following considerations:

- Your model must use a system target file derived from the ERT or GRT target files.
- When you generate code from a model that references other models, configure the top-level model and the referenced models for the same system target file.
- Real-Time Workshop® builds and Embedded IDE Link™ MU software projects do not support external mode in model reference. If you select the external mode option, it is ignored during code generation.
- Your TMF must support use of the shared utilities directory, as described in Supporting Shared Utility Directories in the Build Process in the Real-Time Workshop® documentation.

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

If you start with a model that was created before MATLAB® release R14SP3, use the following command to make your model compatible with model reference :

```
set_param(bdroot,'ModelReferenceCompliant','on') % Sets the Model Referer
```

Code that you generate from Simulink® models by using Embedded IDE Link™ MU software includes the model reference capability. You do not need to set the flag.

4

Verification

What Is Verification? (p. 4-2)	Explains code verification and the features Embedded IDE Link™ MU provides for verifying your code
Using Processor in the Loop (p. 4-3)	Introduces PIL in Embedded IDE Link™ MU
Real-Time Execution Profiling (p. 4-12)	Discusses execution profiling

What Is Verification?

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

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

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

Using Processor in the Loop

In this section...

"Processor-in-the-Loop Overview" on page 4-3

"PIL Block" on page 4-6

"PIL Issues" on page 4-6

"Creating and Using PIL Blocks" on page 4-9

Processor-in-the-Loop Overview

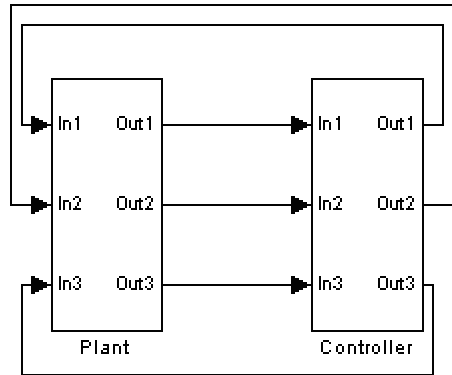
Processor-in-the-loop (PIL) operation provides a powerful verification capability in your development process. Processor-in-the-loop (PIL) cosimulation is a technique designed to help you evaluate how well a candidate algorithm, such as a control system, operates on the actual processor selected for the application.

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 processor hardware.

During the Real-Time Workshop® Embedded Coder™ code generation process, you can create a PIL block from one of several Simulink® components including a model, a subsystem in a model, or subsystem in a library. You then place the generated PIL block inside a Simulink® model that serves as the test harness and run tests to evaluate the processor-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 a classic closed-loop simulation, Simulink® and Stateflow® model such a system as two subsystems with the signals transmitted between them, as shown in the following block diagram:



Your starting point in developing a combined plant and controller system model is to model the combined 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 controller system separately from the plant.

However, these simulation techniques do not help you account for restrictions and requirements imposed by the hardware, such as limited memory resources, or behavior of processor-specific optimized code. When you reach the stage of deploying controller code on the processor hardware, you may need to make extensive adjustments to the controller system to account for the hardware specifics. After you make these adjustments, your deployed system may have diverged significantly from your original model. Such discrepancies can create difficulties if you need to change the original model.

PIL cosimulation addresses these issues by providing an intermediate stage between simulation and deployment. In a PIL cosimulation, the processor participates fully in the simulation loop—hence the term *processor-in-the-loop*.

Two new terms appear in the following sections

- **PIL Algorithm** — The algorithmic code, such as the control algorithm, to test 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 run on the processor. The PIL application is created by linking the PIL algorithm object code with wrapper code or a 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 processor.

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® software generates an executable application for the PIL algorithm. This code runs (in simulated time) on a processor 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 (`out`*n* of the plant) to the processor platform via Green Hills MULTI®. When the processor platform receives signals from the plant model, it executes the PIL algorithm for one sample step. The PIL algorithm returns its output signals (`ont`*n* of the algorithm) computed during this step to Simulink® in `in`*n*, via the Green Hills MULTI® interface. 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 simulation 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.

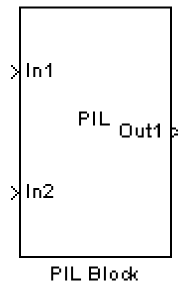
PIL Block

The PIL cosimulation block is the Simulink® block interface to PIL and the interface between the Simulink® plant model and the executable application running on the processor. 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 enables you to perform these operations:

- Select a PIL algorithm
- Build and download a PIL application
- Run a PIL simulation

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



PIL Issues

- "Data Types Must Be the Same Size in MATLAB® and on the Processor" on page 4-7
- "Buses and MUX Signals Not Supported at PIL Component Boundary" on page 4-8

- “Signals with Custom Storage Classes Not Supported at PIL Component Boundary” on page 4-8
- “Continuous Sample Times Not Supported” on page 4-8
- “Real-Time Workshop® grt.tlc-Based Targets Not Supported” on page 4-8
- “Using Breakpoints and PIL” on page 4-8

Consider the following issues when you work with PIL blocks.

Data Types Must Be the Same Size in MATLAB® and on the Processor

Only data types with the same size on the host and processor are supported at the PIL I/O boundary.

The data types used at the PIL I/O boundary are restricted based on the following rule: PIL supports the data type only if the data type size in MATLAB® is the same as the data type size on the processor.

- For `Boolean`, `uint8`, and `int8`, the size is 8 bits on the processor and in MATLAB®.
- For `uint16` and `int16`, the size is 16 bits on processor and in MATLAB®.
- For `uint32` and `int32`, the size is 32 bits on the processor and in MATLAB®.
- For `single`, the size is 32 bits on the processor and in MATLAB®.
- For `double`, the size is 64 bits on the processor and in MATLAB®.

Examples of unsupported data types:

- On the DSP563xx — `single` and `double` are not supported (floating point types are 24 bits on the processor)
- On the 8051 — `double` is not supported (`double` is 32 bits, the same as `single`)

To avoid data type problems, do not use the example data types in your model because the data type on the processor does not match the built-in MATLAB® data type.

Buses and MUX Signals Not Supported at PIL Component Boundary

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

There is no resolution for this issue.

Signals with Custom Storage Classes Not Supported at PIL Component Boundary

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

There is no resolution for this issue.

PIL does support the standard storage classes, such as ExportedGlobal.

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.
```

You must use discrete sample times in your model configuration parameters when you use PIL.

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

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

To use PIL, select the Real-Time Workshop® `multilink.ert` target provided by Real-Time Workshop® Embedded Coder™.

Using Breakpoints and PIL

Green Hills® MULTI® debugger allows you to add breakpoints to your projects. When you run a PIL simulation that includes breakpoints that you added, the following dialog box appears.

Creating and Using PIL Blocks

Using PIL and PIL blocks to verify your processes begins with a Simulink® model of your process. To see an example of one such model used to implement PIL, refer to the demo Comparing Simulation and Processor Implementation with Processor-in-the Loop (PIL) (`multilinkpilsumdiff.mdl`) in the Code Generation Workflow demo for Embedded IDE Link™ MU.

Note Models can have multiple PIL blocks for different subsystems. You cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and incorrect results because the PIL blocks try to run the same code.

To create and use a PIL block

Perform the following tasks to create a new PIL block and use the block in a model:

1. Develop the model of the process to simulate.

   Use Simulink® to build a model of the process to simulate. The blocks in the library `multilinklib` can help you set up the timing and scheduling for your model.

   For information about building Simulink® models, refer to *Getting Started with Simulink®* in the online Help system.

2. Convert your process to a masked subsystem in your model.

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

3. Open the new masked subsystem and add a target preferences block to the subsystem.

   The block library `multilinklib` contains the Target Preferences block to add to your system. Configure the Target Preferences block for your processor. For details about the options on the Target Preferences block, refer to the Target Preferences block reference in the online Help system.

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

   a From the model menu bar, go to **Simulation > Configuration Parameters** in your model to open the Configuration Parameters dialog box.

   b Choose **Real-Time Workshop** from the **Select** tree. Set the configuration parameters for your model as required by the software. Get more information about setting the Real-Time Workshop® parameters in Setting Real-Time Workshop® Options for supported hardware in the online Help system.

   c Under **Target selection**, set the **System target file** to `multilink_ert.tlc` (PIL requires Real-Time Workshop® Embedded Coder™).

5 Configure the model to perform PIL building and PIL block creation.

   a Select `Embedded IDE Link MU` on the **Select** tree.

   b From the **Build Action** list, select `Create_processor_in_the_loop_project` to enable PIL block creation and cosimulation.

   c From the **PIL block action** list, select `Create PIL block`.

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

6 To create the PIL block, right-click the masked subsystem in your model and select **Real-Time Workshop > Build Subsystem** from the context menu.

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

   A new model window opens and the new PIL block appears in it.

7 Copy the new PIL block from the new model to your model, either in parallel to your masked subsystem to cosimulate the processes, or replace your subsystem with the PIL block.

To see the PIL block used in parallel to a masked subsystem, refer to the demo Code Generation Workflow in the demos for Embedded IDE Link MU.

8 Click **Simulation > Start** to run the PIL simulation and view the results.

Real-Time Execution Profiling

In this section...
“Overview” on page 4-12
“Profiling Program Execution” on page 4-12

Overview

The real-time execution profile capability in Embedded IDE Link™ MU uses a set of utilities that record, upload, and analyze the execution profile data for synchronous and asynchronous tasks in your generated code.

Note The software does not support profiling on NEC® V850 and Freescale™ MPC7400 processors.

The profiler generates output in the following formats:

- Graphical display that shows task activation, preemption, task resumption, and task completion. All of the data is presented in the form of a MATLAB® graphic with the data presented by model rates and execution time.
- An HTML report that provides statistical data about the execution of each task in the running process.

In combination, the reports provide a detailed analysis of how your code runs on the processor.

Profiling Program Execution

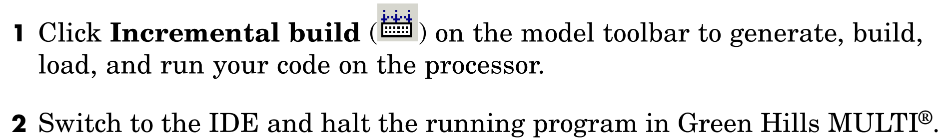
To configure a model to use execution profiling, perform the following steps:

1. Open the Configuration Parameters dialog box for your model.
2. Select `Embedded IDE Link MU` from the **Select** tree.
3. Select **Profile real-time task execution** to enable real-time task profiling.

4. Select **Export IDE link handle to base workspace** and assign a name for the handle in **IDE link handle name**.

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

To run your simulation and then view the execution profile for your model:

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

2. Switch to the IDE and halt the running program in Green Hills MULTI®.

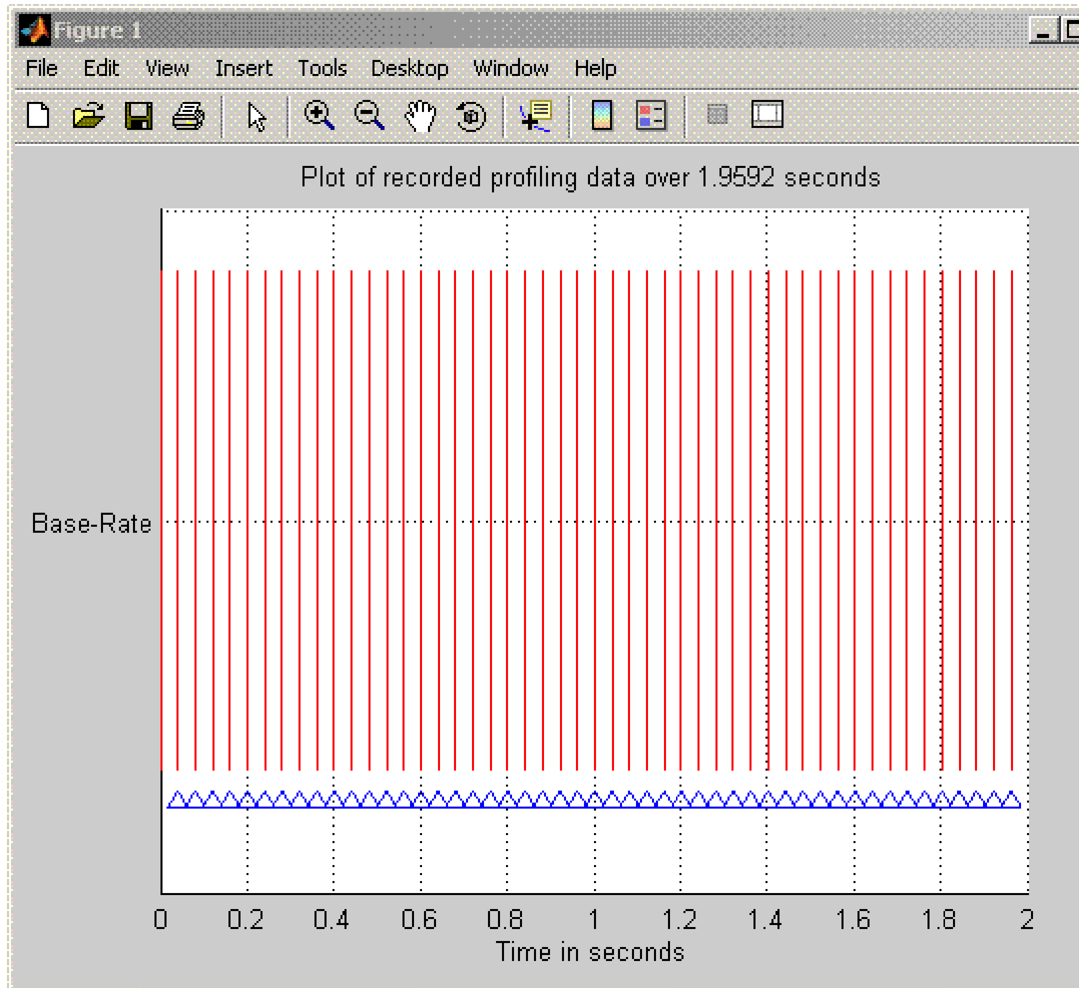
Note Profiling data gathered from a running program may be incorrect.

---

3. To view the profile report, enter the following command at the MATLAB® command prompt:

```
profile(objectname,'report')
```

where *objectname* is the name you provided in **IDE link handle name** in step 4 above and **report** is required to generate the profile report. MATLAB® returns a graphic of the execution report and the HTML execution report. The following figure shows a sample profiling report graphic generated from the Code Generation Workflow demo.



Refer to `profile` for more information.

Functions — By Category

Constructor (p. 5-2)

File and Project Operations (p. 5-3)

Processor Operations (p. 5-4)

Debug Operations (p. 5-5)

Data Manipulation (p. 5-6)

Status Operations (p. 5-7)

Lists the functions and methods available by functional groups

Constructor

`ghsmulti`	Object to communicate with Green Hills MULTI® IDE

File and Project Operations

`activate`	Make specified project active
`add`	Add file or data type to active project
`build`	Build or rebuild current project
`cd`	Set IDE working directory
`close`	Close file in IDE window
`connect`	Connect IDE to processor
`dir`	Files and directories in current IDE window
`getbuildopt`	
`ghsmulticonfig`	Configure Green Hills MULTI®
`info`	Information about processor
`list`	Information listings from MULTI
`new`	New text, project, or configuration file
`open`	Open specified file
`remove`	Remove file from active project in IDE window
`setbuildopt`	Set active configuration build options

Processor Operations

`halt`	Halt program execution by processor
`load`	Load file into processor
`reset`	Stop program execution and reset processor
`restart`	Restart in IDE
`run`	Execute program loaded on processor

Debug Operations

`delete`	Remove breakpoint
`insert`	Insert breakpoint in file

Data Manipulation

`address`	Return address and memory type of specified symbol
`read`	Read data from processor memory
`regread`	Values from processor registers
`regwrite`	Write data values to registers on processor
`write`	Write data to processor memory block

Status Operations

`isrunning`	Determine whether processor is executing process
`visible`	Visibility of IDE window

Functions — Alphabetical List

activate

Purpose

Make specified project active

Syntax

```
activate(id,'my_project.gpj')
```

Description

`activate(id,'my_project.gpj')` uses handle `id` to activate the project named `my_project.gpj` in the IDE. If `my_project.gpj` does not exist in the IDE, MATLAB® issues an error that explains that the specified project does not exist.

MULTI allows you to have two or more projects with the same name open at the same time, such as `c:\try11\try11.gpj` and `c:\try12\try11.gpj`. If you use the following function to activate the project `try11.gpj` at the command prompt, where you do not provide the full path to the project:

```
activate(id,'try11.gpj')
```

the software cannot tell which project named `try11.gpj` to activate and may not activate the correct one. The following steps describe how the software decides which project to activate.

1. Search the current Green Hills MULTI® IDE directory to find the first project with the specified name. If the search finds the project, Embedded IDE Link™ MU activates the project and returns.

2. If the specified project is not found in the IDE, search the MATLAB® path to find a project with this name. If the search finds the project, Embedded IDE Link™ MU activates the project and returns.

3. If the search cannot find a project with the specified name in the Green Hills MULTI® IDE or on the MATLAB® path, the software returns an error saying it could not find the specified project.

See Also

`new`

`remove`

Purpose Add file or data type to active project

Syntax

```
add(id,'my_file')
```

Description `add(id,'my_file')` adds the file `my_file` to the active project from the current MATLAB® working directory. If you do not have an active project in the IDE, MATLAB returns an error message and does not add the file. You can specify the file by name, if the file is in your MATLAB® or Embedded IDE Link™ MU working directory, or provide the fully qualified path to the file when the file is not in your working directories.

To add a file `add.txt` that is in your MATLAB® working directory to the IDE, use the following command:

```
add(id,'add.txt');
```

where `id` is the handle for your `multilink` object. If the file `add.txt` is not in either working directory, the command changes to include the full path to the file:

```
add(id,'fullpathtofile\add.txt');
```

You can add files of all types that the IDE supports. The following table shows the supported file types.

Support File Type	File Extension
C/C++ source files	*.cpp, *.c, *.cxx, *.h, *.hpp, *.hxx
Assembly source files	*.asm, *.dsp
Object and Library files	*.doj, *.dlb
Linker Command files	*.ldf
Green Hills MULTI® support file	*.vdk

See Also `activate`

`cd`

`open`

`remove`

Purpose

Return address and memory type of specified symbol

Syntax

```
a=address(id,'symbolstring')
a=address(id,'symbolstring','scope')
```

Description

`a=address(id,'symbolstring')` returns the address and memory type values for the symbol identified by `symbolstring`. `address` returns the variable in the current (or local) scope. For `address` to work, `symbolstring` must be a symbol in the symbol table for your active project. There must be a linker command file (`lcf`) in your project. If `address` does not find the specified symbol, `a` is empty and MATLAB® software returns a warning message. You can use `address` only after you load the program file.

`a` is a two-element array composed of the symbol address offset and page—`a(1)` is the address offset and `a(2)` is the page. `read` and `write` accept `a` as address inputs.

`a=address(id,'symbolstring','scope')` adds the input argument `scope` that tells the address method whether the symbol is local or global. `Scope` accepts one of the following strings:

string	Description
`global`	Indicates that `symbolstring` represents a global variable
`local`	Indicates that `symbolstring` represents a local variable

Use `local` when the current program scope is the desired scope of the function.

Example

Use `address` to return the address and page of an array named `coef`.

```
a=address(id,'coef')
```

You can use `address` as input for `read` and `write`. This example uses `read` to access the first five elements of the array stored at the address of the global variable `coef`. Use `write` in a similar way.

```
coefvalues=read(id,address(id,'coef','global'),'int32,5)
```

See Also

`load`

`read`

`write`

Purpose

Build or rebuild current project

Syntax

```
build(id)
build(id,timeout)
build(id,'all')
build(id,'all',timeout)
```

Description

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

`build(id,timeout)` incrementally builds the active project with a time limit for how long MATLAB®waits for the build process to complete. `timeout` defines the upper limit in seconds for the period the `build` routine waits for confirmation that the build process is finished. If the build process exceeds the timeout period, control returns toMATLAB® immediately with a timeout error. Usually, `build` causes the processor to initiate a restart, even if it reaches the timeout limit. The timeout error in MATLAB® indicates that confirmation was not received before the timeout period expired. The build action continues. Generally, the build and link process finishes successfully in spite of the timeout error.

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

`build(id,'all',timeout)` rebuilds all the files in the active project applying the timeout limit on how long MATLAB® waits for the build process to complete.

See Also

`isrunning`

`open`

Purpose

Set IDE working directory

Syntax

```
wd = cd(id)
cd (id,'directory')
```

Description

`wd = cd(id)` returns the current IDE working directory, where `id` is a `ghsmulti` object that refers to the Green Hills MULTI® window, or a vector of objects.

`cd (id,'directory')` sets the IDE working directory to `'directory'`. `'directory'` can be a path string relative to your current working directory, or an absolute path. The intended directory must exist. `cd` does not create a new directory. Setting the IDE directory does not affect your MATLAB® working directory.

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

See Also

`dir`

`load`

`open`

Purpose

Close file in IDE window

Syntax

```
close (id,'filename','filetype')
```

Description

`close (id,'filename','filetype')` closes the file named `'filename'` in the active project in the `id` IDE window. If `filename` is not an open file in the IDE, MATLAB® returns a warning message. When you enter null value `[]` for `filename`, close closes the current active file in the IDE. `filename` must match exactly the name of the file to close. If you enter `all` for the filename, `close` closes all files in the project that are of the type specified by `filetype`.

Note CLOSE does not save the file before closing it and it does not prompt you to save the file. You lose any changes you made after the most-recent `save` operation. Use `save` to ensure that your changes are preserved before you close the file.

The parameter `'filetype'` is optional, with the default value of `'text'`. Allowed `'filetype'` strings are `'project'`, `'projectgroup'`, `'text'`, and `'workspace'`. Here are some examples of `close` operation commands. In these examples, `id` is a `ghsmulti` object handle to the IDE.

`close(id,'all','project')` — Closes all open project files

`close(id,'my.gpj','project')` — Closes the open project `my.gpj`

`close(id,[],'project')` — Closes the active open project

`close(id,'all','projectgroup')` — Close all open project groups.

`close(id,'myg.dpg','projectgroup')` — Closes the project group: `myg.dpg`

`close(id,[],'projectgroup')` — Closes the active project group

`close(id,'all','text')` — Close all text files

`close(id,'text.c','text')` — Closes the text file `text.c`

`close(id,[],'text')` — Closes the active text file

See Also

`add`

`open`

`save`

Purpose

Connect IDE to processor

Syntax

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

Decription

`connect(id)` connects the IDE to the processor hardware or simulator. `id` is the `ghsmulti` object that accesses the IDE.

`connect(id,debugconnection)` connects the IDE to the processor using the debug connection you specify in `debugconnection`. Enter `debugconnection` as a string enclosed in single quotation marks. `id` is the `ghsmulti` object that references the IDE. Refer to Examples to see this syntax in use.

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

Example

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

```
connect(id,'simppc -fast -dec -rom_use_entry -cpu=ppc5554')
```

See Also

`load`

`run`

delete

Purpose Remove breakpoint

Syntax

```
delete(id,addr)
delete(id,'filename','linenumber')
delete(id,'all')
```

Description

`delete(id,addr)` removes a breakpoint located at the memory address `addr` of the processor. Provide the address input value in hexadecimal format, such as `0x244fc`, or `0x0014`.

`delete(id,'filename','linenumber')` removes the breakpoint located at the line number `'linenumber'` in the file `'filename'` for the processor.

`delete(id,'all')` removes all breakpoints in the current project source files.

See Also `insert`

Purpose

Files and directories in current IDE window

Syntax

```
dir(id)
d=dir(id)
```

Description

`dir(id)` lists the files and directories in the IDE working directory, where `id` is the object that references the IDE. `id` can be either a single handle, or a vector of handles. When `id` is a vector, `dir` returns the files and directories for each handle.

`d=dir(id)` returns the list of files and directories as an M-by-1 structure in `d` with the following fields for each file and directory, as shown in the following table.

Field Name	Description
`name`	Name of the file or directory.
`date`	Date of most recent file or directory modification.
`bytes`	Size of the file in bytes. Directories return 0 for the number of bytes.
`isdirectory`	0 if this is a file, 1 if this is a directory.

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

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

See Also

`cd`

`open`

display

Purpose

Properties of ghsmulti object

Syntax

```
display(id)
```

Description

display(id) displays the properties and property values of the ghsmulti object id.

For example, when you create id associated with localhost and port number 4444, display(id) returns the following information in the MATLAB® command window:

```
display(id)

MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\v800\
```

See Also

get in the MATLAB® Function Reference

Syntax

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

Description

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

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

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

Purpose

Object to communicate with Green Hills MULTI® IDE

Syntax

```
id = ghsmulti
id = ghsmulti('propertyname1',propertyvalue1,'propertyname2',
   propertyvalue2,...,'timeout',value)
```

Description

`id = ghsmulti` returns object `id` that communicates with a target processor. Before you use this command for the first time, use `ghsmulticonfig` to configure your MULTI software installation to identify the location of your MULTI software, your processor configuration, your debug server and the host name and port number of the Embedded IDE Link™ MU service.

`ghsmulti` creates an interface between MATLAB® and Green Hills MULTI. If this is the first time you have used `ghsmulti`, you must supply the properties and property values shown in following table as input arguments:

Property Name	Description
hostname	Specifies the name of the machine hosting the Embedded IDE Link™ MU service. The default host name is localhost indicating that the service is on the local PC. Replace localhost with the name you entered in **Host name** on the Embedded IDE Link™ MU Configuration dialog box.
portnum	Specifies the port to connect to the Embedded IDE Link™ MU service on the host machine. The default value for portnum is 4444. Replace portnum with the number you entered in **Port number** on the Embedded IDE Link™ MU Configuration dialog box.

When you invoke ghsmulti, it starts the Embedded IDE Link™ MU service. If you selected the **Show server status window** option on the Embedded IDE Link™ MU Configuration dialog box (refer to ghsmulticonfig) when you configured your MULTI installation, the service appears in your Microsoft Windows task bar. If you clear **Show server status window**, the service does not appear.

Parameters that you pass as input arguments to ghsmulti are interpreted as object property definitions. Each property definition consists of a property name followed by the desired property value (often called a PV, or property name/property value, pair).

Note The output (left-hand argument) object name you provide for ghsmulti cannot begin the an underscore, such as _id.

`id = ghsmulti('hostname','name','portnum','number',...)` returns a `ghsmulti` object `id` that you use to interact with a processor in the IDE from the MATLAB® command prompt. If you enter a `hostname` or `portnum` that are not the same as the ones you provided when you configured your MULTI installation, Embedded IDE Link™ MU returns an error that it could not connect to the specified host and port and does not create the object.

You use the debugging methods (refer to "Debug Operations" on page 5-5 for the methods available) with this object to access memory and control the execution of the processor. `ghsmulti` also enables you to create an array of objects for a multiprocessor board, where each object refers to one processor on the board. When `id` is an array of objects, any method called with `id` as an input argument is sent sequentially to all processors connected to the `ghsmulti` object. Green Hills MULTI® provides the communication between the IDE and the processor.

After you build the `ghsmulti` object `id`, you can review the object property values with `get`, but you cannot modify the `hostname` and `portnum` property values. You can use set to change the value of other properties.

`id = ghsmulti('propertyname1',propertyvalue1,'propertyname2',propertyvalue2,...` sets the global time-out value in seconds to `value` in `id`. MATLAB® waits for the specified time-out period to get a response from the IDE application. If the IDE does not respond within the allotted time-out period, MATLAB® exits from the evaluation of this function.

Examples

This example demonstrates `ghsmulti` using default values.

`id = ghsmulti('hostname','localhost','portnum',4444);`

returns a handle to the default host and port number—`localhost` and `4444`.

```
id=ghsmulti('hostname','localhost','portnum',4444)

MULTI Object:
```

```
Host Name        : localhost
Port Num         : 4444
Default timeout  : 10.00 secs
MULTI Dir        : C:\ghs\multi500\ppc\
```

See Also

ghsmulticonfig

ghsmulticonfig

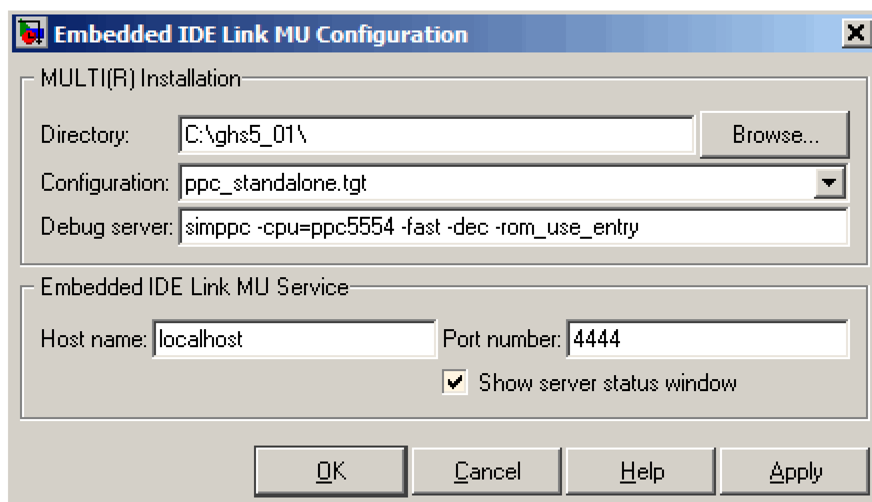
Purpose Configure Green Hills MULTI®

Syntax `ghsmulticonfig`

Description `ghsmulticonfig` launches the Embedded IDE Link™ MU Configuration dialog box that you use to configure your Embedded IDE Link™ MU installation to work with MULTI.

Note The Embedded IDE Link™ MU Configuration dialog box is the only place you set the host name and port number configuration.

The dialog box, shown in the following figure, provides controls that specify parameters such as where you installed MULTI and the name of the host machine to use.



Directory

Tells Embedded IDE Link™ MU the path to your Green Hills MULTI software installation. Enter the full path to the Green

Hills® MULTI® executable, `multi.exe`, in your installation. To search for the executable file, click **Browse**.

If you have more than one version of MULTI, such as PowerPC (`ppc`) and V800 (`v800`), specify the path to `multi.exe` in the processor-specific version to use.

If you do not provide or select a correct path to the executable file, Embedded IDE Link™ MU ignores your entry and returns an error message saying it could not find the executable `multi.exe` in the specified or selected directory.

Configuration

Specifies the primary processor family to use to develop your projects in MULTI. This corresponds to a `.tgt` file you select before you can download and execute code. Select your family file from the list. In many cases, the *`family`*`_standalone.tgt` option is the appropriate choice. For example, if you develop on the Freescale™ MPC5xx, you could select `ppc_standalone.tgt`. Embedded IDE Link™ MU stores your selection. You do not need to repeat this setup task unless you change processors.

Debug server

Like the primary target configuration, MULTI needs a debug connection. This parameter enables you to enter the name of your debug connection. Embedded IDE Link™ MU uses this connection to specify options about the processor, such as processor to use, board support library, and processor endianness. For more information about the Debug server, refer to your Green Hills MULTI documentation.

For example, if you are using the Freescale MPC5554 simulator, you could enter the string `simppc -cpu=ppc5554 -dec -rom_use_entry`. Valid strings for specifying simulators in **Debug server** appear in the following table.

Processor	Type	Configuration	Debug Server Parameter String
MPC5554	Simulator	ppc_standalone.tgt	simppc -cpu=ppc5554 -dec -rom_use_entry
MPC7400	Simulator	ppc_standalone.tgt	simppc -cpu=7400
BlackFin 537	Simulator	bf_standalone.tgt	simbf -cpu=bf537 -fast
NEC V850	Simulator	v800_standalone.tgt	sim850 -cpu=v850

For information about using hardware in your development work, refer to *Connecting to Your Target* in the MULTI documentation. The string you specify for **Debug server** can be the command or the name of the connection if you have one configured in the Connection Organizer in MULTI.

Host name
Specify the name of the machine that runs the Embedded IDE Link™ MU service. Enter `localhost` if the service runs on your PC. `localhost` is the only supported host name.

Port number
Specify the port the Embedded IDE Link™ MU service uses to communicate with MULTI. The default port number is 4444. If you change the port value, verify that the port is available for use. If the port you assign is not available, Embedded IDE Link™ MU returns an error when you try to create a `ghsmulti` object.

Show server status window
Select this option to display the Embedded IDE Link™ MU service status in the Microsoft Windows Task bar. Clearing the option removes the service from the task bar. Best practice is to select this option. Keeping this option selected enables the software to shut down the communication services for Green Hills® MULTI® completely.

Purpose

Halt program execution by processor

Syntax

```
halt(id)
halt(id,timeout)
```

Description

`halt(id)` stops the program running on the processor. After you issue this command, MATLAB® waits for a response from the processor that the processor has stopped. By default, the wait time is 10 seconds. If 10 seconds elapses before the response arrives, MATLAB® returns an error. In this syntax, the timeout period defaults to the global timeout period specified in `id`. Use `get(id)` to determine the global timeout period. However, the processor usually stops in spite of the error message.

To resume processing after you halt the processor, use `run`. Also, the `read(id,'pc')` function can determine the memory address where the processor stopped after you use `halt`

`halt(id,timeout)` immediately stops program execution by the processor. After the processor stops, `halt` returns to the host. `timeout` defines, in seconds, how long the host waits for the processor to stop running.

`timeout` defines the maximum time the routine waits for the processor to stop. If the processor does not stop within the specified timeout period, the routine returns with a timeout error.

Examples

Use one of the provided demonstration programs to show how halt works. From the Green Hills MULTI® demonstration programs, load and run one of the demonstration projects.

At the MATLAB® prompt, create an object that refers to Green Hills MULTI®

```
id = ghsmulti
```

Check whether the program is running on the processor.

```
isrunning(id)
```

```
ans =

     1

id.isrunning % Alternate syntax for checking the run status.

ans =

     1
halt(id) % Stop the running application on the processor.
isrunning(id)

ans =

     0
```

Issuing the halt stops the process on the processor. Checking in Green Hills MULTI® confirms that the process has stopped.

See Also

isrunning

reset

run

Purpose Information about processor

Syntax

```
iid = info(id)
```

Description

`iid = info(id)` returns property names and property values associated with the debugger and processor referred to by `id`. `iid` is a structure containing the information elements and values shown in the following table:

Structure Element	Data Type	Description
`iid.CurBrkPt`	String	When the debugger is stopped at a breakpoint, the field reports the index of the breakpoint. Otherwise, this value is-1.
`iid.File`	String	Name of the current file shown in the debugger source pane.
`iid.Line`	Integer	Line number of the cursor position in the file in the debugger source pane. If no file is open in the source pane, this value is -1
`iid.MultiDir`	String	Full path to your Green Hills® MULTI® installation the root directory). For example `'C:\ghs5_01'`
`iid.PID`	Double	Process ID from the debug server in MULTI.
`iid.Procedure`	String	Current procedure in the debugger source pane.
`iid.Process`	Double	Program number, defined by MULTI, of the current program.
`iid.Remote`	String	Status of the remote connection, either `Connected` or `Not connected`.
`iid.Selection`	String	The string highlighted in the debugger. If there is no string highlighted, this value is '`null`'.

Structure Element	Data Type	Description
iid.State	String	State of the loaded program. The possible reported states appear in the following list: • About to resume • Dying • Just executed • Just forked • No child • Running • Stopped • Zombied For details about the states and their definitions, refer to your Green Hills® MULTI® debugger documentation.
iid.Target	Double	Unique identifier the indicates the processor family and variant.
iid.TargetOS	Double	Real-time operating system on the processor if one exists. Provides both the major and minor revision information.
iid.TargetSeries	Double	Whether the processor belongs to a series of processors. For details about the processor series, refer to your Green Hills® MULTI® debugger documentation.

info returns valid information when the IDE debugger is connected to processor hardware or a simulator.

Using info with multiprocessor boards

Method info works with targets that have more than one processor by returning the information for each processor accessed by the id object you created with ghsmulti. The structure of information returned is identical to the single processor case, for every included processor.

Examples

On a PC with a simulator configured in MULTI, info returns the following configuration information after stopping a running simulation:

```
iid=info(test_obj1)

iid =

        CurBrkPt: 0
            File: 'C:\Work\Adaweb\matlab\Compute_Sum_and_Diff_multi
            Line: 3
        MultiDir: 'C:\ghs5_01'
             PID: 2380
       Procedure: 'main'
         Process: 0
          Remote: 'Connected'
       Selection: '(null)'
           State: 'Stopped'
          Target: 4325392
        TargetOS: [2x1 double]
    TargetSeries: 3
```

When you create a new ghsmulti object, the response from info looks like the following before you load a project.

```
iid=info(test_obj2)

test_obj2 =

        CurBrkPt: []
            File: []
            Line: []
```

```
       MultiDir: []
            PID: []
      Procedure: []
        Process: []
         Remote: []
      Selection: []
          State: []
         Target: []
       TargetOS: []
   TargetSeries: []
```

See Also

ghsmulti, dec2hex, get, set

Purpose

Insert breakpoint in file

Syntax

```
insert(id,addr)
insert(id,'filename','linenumber')
```

Description

`insert(id,addr)` inserts a breakpoint at the memory address specified by the `addr` parameter .`id` identifies the session that adds the breakpoint.

`insert(id,'filename','linenumber')` inserts a breakpoint at the line `'linenumber'` in the file `'filename'`.

See Also

`address`

`delete`

`run`

isrunning

Purpose

Determine whether processor is executing process

Syntax

```
isrunning(id)
```

Description

`isrunning(id)` returns `1` when the processor is executing a program. When the processor is halted, `isrunning` returns 0.

Examples

`isrunning` lets you determine whether the processor is running. After you load a program to the processor, use `isrunning` to verify that the program is running.

```
id=ghsmulti

MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\v800\

visible(id,1)
load(id,'program.dxe','program')
run(id)
isrunning(id)

ans =

     1
halt(id)
isrunning(id)

ans =

     0
```

See Also

`halt`

`load`

```
run
```

list

Purpose

Information listings from MULTI

Syntax

```
list(ff,varname)
infolist = list(id,'type')
infolist = list(id,'type',typename)
```

Description

`list(ff,varname)` lists the local variables associated with the function accessed by function object `ff`. Compare to `list(id,'variable','varname')` which works the same way to return variables from object `id`.

Note `list` does not recognize or return information about variables that you declare in your code but that are not used or initialized.

Some restrictions apply when you use `list` with function objects. `list` generates an error in the following circumstances:

- When `varname` is not a valid input argument for the function accessed by `ff`

  For example, if your function declaration is

  ```
  int foo(int a)
  ```

  but you request information about input argument `b`, which is not defined

  ```
  list(ff,'b')
  ```

  MATLAB® returns an error.

- When `varname` is the same as a variable assigned by MATLAB®. Usually this happens when you use `declare` to pass a function declaration to MATLAB® and the declaration string does not match the declaration for `ff` as determined when you created `ff`.

In an example that demonstrates this problem, the function declaration has a name for the first input, `a`. In the `declare` call, the declaration string does not provide a name for the first input, just a data type, `int`. When you issue the `declare` call, MATLAB® software names the first input `ML_Input1`. If you try to use `list` to get information about the input named `ML_Input`, `list` returns an error. Here is the code, starting with the function declaration in your code:

```
int foo(int a)  % Function declaration in your source code
declare(ff,'decl','int foo(int)')
% MATLAB generates a warning that it has assigned the name
% ML_Input to the first input argument
list(ff,'ML_Input') % list returns an error for this call
```

- When `varname` does not match the input name in the function declaration provided in your source code, as compared to the declaration string you used in a declare operation.

  Assume your source code includes a function declaration for `foo`:

  ```
  int foo(int a);
  ```

  Now pass a declaration for `foo` to MATLAB®:

  ```
  declare(ff,'decl','int foo(int b)')
  ```

  MATLAB® issues a warning that the input names do not match. When you use list on the input argument `b`,

  ```
  list(ff,'b')
  ```

  `list` returns an error.

- When `varname` is an input to a library function. `list` always fails in this case. It does not matter whether you use declare to provide the declaration string for the library function.

Note When you call `list` for a variable in a function object `list(ff,varname)`the `address` field may contain an incorrect address if the program counter is not within the scope of the function that includes `varname` when you call `list`.

`infolist = list(id,type)` reads information about your MULTI project and returns it in *`infolist`*. Different types of information and return formats are possible depending on the input arguments you supply to the `list` function call. The `type` argument specifies which information listing to return. To determine the information that `list` returns, use one of the following as the `type` parameter string:

- **`project`** — Tell `list` to return information about the current project in MULTI.
- **`variable`** — Tell `list` to return information about one or more embedded variables.
- **`globalvar`** — Tell `list` to return information about one or more global embedded variables.
- **`function`** — Tell `list` to return details about one or more functions in your project.
- **`type`** — Tell `list` to return information about one or more defined data types, including `struct`, `enum`, and `union`. C data type typedefs are excluded from the list of data types.

Note, the list function returns dynamic Code Composer information that can be altered by the user. Returned listings represent snapshots of the current Code Composer studio configuration only. Be aware that earlier copies of `infolist` might contain stale information.

Also, `list` may report incorrect information when you make changes to variables from MATLAB®. To report variable information, `list` uses the MULTI API, which only knows about variables in MULTI. Your changes from MATLAB®, such as changing the data type of a variable,

do not appear through the API and `list`. For example, the following operations return incorrect or old data information from `list`.

Suppose your original prototype is

```
unsigned short tgtFunction7(signed short signedShortArray1[]);
```

After creating the `function` object `fcnObj`, perform a `declare` operation with this string to change the declaration:

```
unsigned short tgtFunction7(unsigned short signedShortArray1[]);
```

Now try using `list` to return information about `signedShortArray1`.

```
list(fcnObj,'signedShortArray1')

 address: [3442 1]
location: [1x66 char]
    size: 1
    type: 'short *'
 bitsize: 16
 reftype: 'short'
  referent: [1x1 struct]
member_pts_to_same_struct: 0
      name: 'signedShortArray1'
```

The `type` field reports the original data type `short`.

You get this is because `list` uses the MULTI API to query information about any particular variable. As far as the API is concerned, the first input variable is a `short*`. Changing the declaration does not change anything.

`infolist = list(id,'`**`project`**`')` returns a vector of structures containing project information in the format shown here when you specify option `type` as **`project`**.

infolist Structure Element	Description
`infolist(1).name`	Project file name (with path).
`infolist(1).type`	Project type — `project`,`projlib`, or `projext`, refer to `new`
`infolist(1).targettype`	String description of target CPU
`infolist(1).srcfiles`	Vector of structures that describes project source files. Each structure contains the name and path for each source file — `infolist(1).srcfiles.name`
`infolist(1).buildcfg`	Vector of structures that describe build configurations, each with the following entries: • `infolist(1).buildcfg.name` — the build configuration name • `infolist(1).buildcfg.outpath` — the default directory for storing the build output.
`infolist(2)....`	...
`infolist(n)....`	...

`infolist = list(id,'`**`variable`**`')` returns a structure of structures that contains information on all local variables within scope. The list also includes information on all global variables. Note, however, that if a local variable has the same symbol name as a global variable, list returns the information about the local variable.

`infolist = list(id,'`**`variable`**`',varname)` returns information about the specified variable `varname`.

`infolist = list(id,'`**`variable`**`',varnamelist)` returns information about variables in a list specified by `varnamelist`. The information

returned in each structure follows the format below when you specify option `type` as **`variable`**:

infolist Structure Element	Description
`infolist.varname(1).name`	Symbol name.
`infolist.varname(1).isglobal`	Indicates whether symbol is global or local.
`infolist.varname(1).location`	Information about the location of the symbol.
`infolist.varname(1).size`	Size per dimension.
`infolist.varname(1).uclass`	ghsmulti object class that matches the type of this symbol.
`infolist.varname(1).bitsize`	Size in bits. More information is added to the structure depending on the symbol type.
`infolist.(varname1).type`	Data type of symbol.
`infolist.varname(2)....`	...
`infolist.varname(n)....`	...

`list` uses the variable name as the field name to refer to the structure information for the variable.

`infolist = list(id,'`**`globalvar`**`')` returns a structure that contains information on all global variables.

`infolist = list(id,'`**`globalvar`**`',varname)` returns a structure that contains information on the specified global variable.

`infolist = list(id,'`**`globalvar`**`',varnamelist)` returns a structure that contains information on global variables in the list. The returned information follows the same format as the syntax `infolist = list(id,'variable',...)`.

`infolist = list(id,'`**`function`**`')` returns a structure that contains information on all functions in the embedded program.

`infolist = list(id,'function',functionname)` returns a structure that contains information on the specified function `functionname`.

`infolist = list(id,'function',functionnamelist)` returns a structure that contains information on the specified functions in `functionnamelist`. The returned information follows the format below when you specify option `type` as **`function`**:

infolist Structure Element	Description
`infolist.functionname(1).name`	Function name
`infolist.functionname(1).filename`	Name of file where function is defined
`infolist.functionname(1).address`	Relevant address information such as start address and end address
`infolist.functionname(1).funcvar`	Variables local to the function
`infolist.functionname(1).uclass`	`ghsmulti` object class that matches the type of this symbol — **`function`**
`infolist.functionname(1).funcdecl`	Function declaration — where information such as the function return type is contained
`infolist.functionname(1).islibfunc`	Is this a library function?
`infolist.functionname(1).linepos`	Start and end line positions of function

infolist Structure Element	Description
`infolist.functionname(1).funcinfo`	Miscellaneous information about the function
`infolist.functionname(2)...`	...
`infolist.functionname(n)...`	...

To refer to the function structure information, `list` uses the function name as the field name.

`infolist = list(id,'type')` returns a structure that contains information on all defined data types in the embedded program. This method includes struct, enum and union data types and excludes typedefs. The name of a defined type is its C struct tag, enum tag or union tag. If the C tag is not defined, it is referred to by the MULTI compiler as `'$fake`*n*`'` where *n* is an assigned number.

`infolist = list(id,'type',typename)` returns a structure that contains information on the specified defined data type.

`infolist = list(id,'type',typenamelist)` returns a structure that contains information on the specified defined data types in the list. The returned information follows the format below when you specify option `type` as **`type`**:

infolist Structure Element	Description
`infolist.typename(1).type`	Type name
`infolist.typename(1).size`	Size of this type
`infolist.typename(1).uclass`	ghsmulti object class that matches the type of this symbol. Additional information is added depending on the type

infolist Structure Element	Description
infolist.typename(2)....	...
infolist.typename(n)....	...

For the field name, list uses the type name to refer to the type structure information.

The following list provides important information about variable and field names:

- When a variable name, type name, or function name is not a valid MATLAB® structure field name, list replaces or modifies the name so it becomes valid.
- In field names that contain the invalid dollar character $, list replaces the $ with DOLLAR.
- Changing the MATLAB® field name does not change the name of the embedded symbol or type.

Examples

This first example shows list used with a variable, providing information about the variable varname. Notice that the invalid field name _with_underscore gets changed to Q_with_underscore. To make the invalid name valid, list inserts the character Q before the name.

```
varname1 = '_with_underscore'; % invalid fieldname
list(id,'variable',varname1);
ans =

    Q_with_underscore : [varinfo]
ans. Q_with_underscore
ans=

        name: '_with_underscore'
    isglobal: 0
    location: [1x62 char]
        size: 1
```

```
      uclass: 'numeric'
        type: 'int'
     bitsize: 16
```

To demonstrate using `list` with a defined C type, variable `typename1` includes the `type` argument. Because valid field names cannot contain the `$` character, `list` changes the `$` to `DOLLAR`.

```
typename1 = '$fake3'; % name of defined C type with no tag
list(id,'type',typename1);
ans =

         DOLLARfake0 : [typeinfo]

ans.DOLLARfake0=

          type: 'struct $fake0'
          size: 1
       uclass: 'structure'
       sizeof: 1
      members: [1x1 struct]
```

When you request information about a project in MULTI, you see a listing like the following that includes structures containing details about your project.

```
projectinfo=list(id,'project')

projectinfo =

          name: 'D:\Work\c6711dskafxr_c6000_rtw\c6711dskafxr.pjt'
          type: 'project'
    targettype: 'TMS320C67XX'
      srcfiles: [69x1 struct]
      buildcfg: [3x1 struct]
```

See Also

`info`

load

Purpose

Load file into processor

Syntax

```
load(id,'filename',timeout)
load(  ,timeout)
```

Description

`load(id,'`*`filename`*`',timeout)` transfers file `'my_file.dxe'` to the processor. *`filename`* can include a full path to the file, or the name of a file that is in the current working directory of Green Hills MULTI®. Use the function `cd` to check or modify the Green Hills MULTI®working directory. Use this function only with program files that you created by a Green Hills MULTI® build process. When you issue the `load` command, the command waits for the period defined by `timeout` in `id` for the process to complete—ten seconds.

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

See Also

`cd`

`dir`

`open`

Purpose

New text, project, or configuration file

Syntax

`new(id,'name','type')`

Description

`new(id,'name','type')` creates a new file, project, or build configuration in the active project. Input argument `name` specifies the name assigned to identify the new file, project, or configuration.

When you are creating a new executable project or library project, *`name`* is a filename that can include the full path to the new file. If you omit the path, `new` creates the new file or project in your current Green Hills MULTI® working directory.

If your name input argument does not include the file extension, and you do not include the `type` argument, `new` creates a new executable project in the IDE with the `gpj` extension.

To define the kind of entity to create, *`type`* accepts the strings shown in the following table.

Type String	Description
`project`	Create a new MULTI executable project in the current IDE instance. Sometimes this is called a *DSP executable file*.
`projectlib`	Create a new MULTI library project in the current IDE instance.

Examples

`new(id,'my_project.gpj','project')` creates a new project `'my_project.gpj'` of type `project`.

The '`project`' argument is optional; the default project type is an executable project. When you include the `gpj` extension on the name of the new project `my_project.gpj`, `new` automatically creates a project file.

`new(id,'my_library_project','projectlib')` creates a new library project in the IDE instance that `id` references. To create the library project, you must include the `'projectlib'` input argument.

See Also

`activate`

`close`

`save`

Purpose

Open specified file

Syntax

```
open(id,'filename')
open( ,'filetype')
open( ,timeout)
```

Description

`open(id,'filename')` opens file `filename` in the IDE. If you specify the file extension in `filename`, `open` opens the file of that type. If you omit the file extension from the name, `open` assumes the file to open is a project. Files that do not have the `.gpj` extension or do not have an extension are assumed to be projects. The following table presents the possible file types and extensions.

Extension	Assumed File Type	Description
`txt`, `.c`, `.asm`, `.cpp`, `.h`, and all file extensions not listed elsewhere in this table	**`text`**	Treated as text file
`gpj` or no extension	**`project`**	Treated as Green Hills® MULTI®project

If the file to open does not exist in the current project or directory path, MATLAB® returns a warning and returns control to MATLAB®.

`open( ,'filetype')` identifies the type of file to open. This can be useful when your project includes files of different types that have the same name or when you want to open a project, project group, or workspace. Using the input argument `filetype` overrides the file type defined by the file extension in the file name. The preceding table defines the valid file type extensions.

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

See Also

`cd`

`dir`

`load`

`new`

Purpose

Read data from processor memory

Syntax

```
mem = read(id,address)
mem = read(...,datatype)
mem = read(...,count)
mem = read(...,memorytype)
mem = read(...,timeout)
```

Description

`mem = read(id,address)` returns a block of data values from the memory space of the DSP processor referenced by `id`. The block to read begins from the DSP memory location given by the input parameter `address`. The data is read starting from `address` without regard to type-alignment boundaries in the DSP. Conversely, the byte ordering defined by the data type is automatically applied.

`address` is a decimal or hexadecimal representation of a memory address in the DSP. In all cases, the full memory address consist of two parts:

- The start address
- The memory type

You can define the memory type value can be explicitly using a numeric vector representation of the address (see below).

Alternatively, the `id` object has a default memory type value that is applied if the memory type value is not explicitly incorporated in the passed address parameter. In DSP processors with only a single memory type, it is possible to specify all addresses using the abbreviated (implied memory type) format by setting the `id` object memory type value to zero.

Note You cannot read data from processor memory while the processor is running.

Provide the address parameter either as a numerical value that is a decimal representation of the DSP memory address, or as a string that `read` converts to the decimal representation of the start address. (Refer to function `hex2dec` in the *MATLAB® Function Reference*. `read` uses `hex2dec` to convert the hexadecimal string to a decimal value).

The examples in the following table demonstrate how `read` uses the `address` parameter:

address Parameter Value	Description
`131082`	Decimal address specification. The memory start address is 131082 and memory type is 0. This is the same as specifying [131082 0].
`[131082 1]`	Decimal address specification. The memory start address is 131082 and memory type is 1.
`'2000A'`	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify `address` as a cell array. You can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myadddress1{2} =
'Program(PM) Memory';
```

```
myaddress2 myaddress2{1} = '20000'; myadddress2{2} =
'Program(PM) Memory';
```

```
myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem = read(...,datatype)` where the input argument `datatype` defines the interpretation of the raw values read from DSP memory. Parameter `datatype` specifies the data format of the raw memory image. The data is read starting from `address` without regard to data type alignment

boundaries in the DSP. The byte ordering defined by the data type is automatically applied. This syntax supports the following MATLAB® data types:

MATLAB® Data Type	Description
double	IEEE double-precision floating point value
single	IEEE single-precision floating point value
uint8	8-bit unsigned binary integer value
uint16	16-bit unsigned binary integer value
uint32	32-bit unsigned binary integer value
int8	8-bit signed two's complement integer value
int16	16-bit signed two's complement integer value
int32	32-bit signed two's complement integer value

`read` does not coerce data type alignment. Some combinations of `address` and `datatype` will be difficult for the processor to use.

`mem = read(...,count)` adds the `count` input parameter that defines the dimensions of the returned data block `mem`. To read a block of multiple data values. Specify `count` to determine how many values to read from `address`. `count` can be a scalar value that causes `read` to return a column vector that has `count` values. You can perform multidimensional reads by passing a vector for `count`. The elements in the input vector of `count` define the dimensions of the returned data matrix. The memory is read in column-major order. `count` defines the

dimensions of the returned data array mem as shown in the following table.

- n — Read n values into a column vector.
- [m,n] — Read m-by-n values into m by n matrix in column-major order.
- [m,n,...] — Read a multidimensional matrix m-by-n-by…of values into an m-by-n-by…array.

To read a block of multiple data values, specify the input argument count that determines how many values to read from address.

mem = read(…,memorytype) adds an optional input argument memorytype. Object id has a default memory type value 0 that read applies if the memory type value is not explicitly incorporated into the passed address parameter.

In processors with only a single memory type, it is possible to specify all addresses using the implied memory type format by setting the id memorytype property value to zero. Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
'program(pm) memory'	0	Blackfin and SHARC
'data(dm) memory'	1	SHARC
'data(dm) short word memory'	2	SHARC
'external data(dm) byte memory'	3	SHARC
'boot(prom) memory'	4	SHARC

`mem = read(...,timeout)` adds the optional parameter *`timeout`* that defines how long, in seconds, MATLAB® waits for the specified read process to complete. If the time-out period expires before the read process returns a completion message, MATLAB® returns an error and returns. Usually the read process works correctly in spite of the error message.

Examples

This example reads one 16–bit integer from memory on the processor.

```
mlvar = read(id,131072,'int16')
```

`131072` is the decimal address of the data to read.

You can read more than one value at a time. This `read` command returns 100 32–bit integers from the address 0x20000 and plots the result in MATLAB®.

```
data = read(id,'20000','int32',100)
plot(double(data))
```

See Also

`write`

regread

Purpose

Values from processor registers

Syntax

```
reg = regread(id,'regname','represent',timeout)
reg = regread(id,'regname','represent')
reg = regread(id,'regname')
```

Description

`reg = regread(id,'regname','represent',timeout)` reads the data value in the `regname` register of the target processor and returns the value in `reg` as a double-precision value. For convenience, `regread` converts each return value to the MATLAB® `double` datatype. Making this conversion lets you manipulate the data in MATLAB®. String `regname` specifies the name of the source register on the target. `ghsmulti` object `id` defines the target to read from. Valid entries for `regname` depend on your target processor.

Note `regread` does not read 64-bit registers, like the `cycle` register on Blackfin processors.

Register names are not case-sensitive — `a0` is the same as `A0`.

For example, the following registers are some of the many available on the MPC5500 processor:

- '`acc`' — Accumulator A register
- `sprg0` through `sprg7` — SPR registers

Note Use `read` (called a direct memory read) to read memory-mapped registers.

The `represent` input argument defines the format of the data stored in `regname`. Input argument `represent` takes one of three input strings:

represent String	Description
`2scomp`	Source register contains a signed integer value in two's complement format. This is the default setting when you omit the `represent` argument.
`binary`	Source register contains an unsigned binary integer.
`ieee`	Source register contains a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are reading from 32 and 64 bit registers on the target.

To limit the time that `regread` spends transferring data from the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the read operation. You might find this useful for limiting prolonged data transfer operations. If you omit the `timeout` option in the syntax, `regread` defaults to the global time-out defined in `id`.

`reg = regread(id,'regname','represent')` does not set the global time-out value. The time-out value in `id` applies.

`reg = regread(id,'regname')` does not define the format of the data in `regname`.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations at any time during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables during intermediate times in program operation may not get reflected in the register.

This is true for local variables as well.

One way to see this is to write a line of code that uses the variable and see if the result is consistent.

```
register int a = 100;
int b;
 ...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or Embedded IDE Link™ MU.

Examples

For the MPC5554 processor, most registers are memory-mapped and consequently are available using `read` and `write`. However, use `regread` to read the PC register. The following command demonstrates how to read the PC register. To identify the target, `id` is a `ghsmulti` object for MULTI.

```
id.regread('PC','binary')
```

To tell MATLAB® what data type you are reading, the string `binary` indicates that the `PC` register contains a value stored as an unsigned binary integer.

In response, MATLAB® displays

```
ans =

           33824
```

For processors in the Blackfin family, `regread` lets you access processor registers directly. To read the value in general purpose register cycles, type the following function.

```
treg = id.regread('cycles','2scomp');
```

`treg` now contains the two's complement representation of the value in `A0`.

See Also

`read`, `regwrite`, `write`

regwrite

Purpose

Write data values to registers on processor

Syntax

```
regwrite(id,'regname',value,'represent',timeout)
regwrite(id,'regname',value,'represent')
regwrite(id,'regname',value,)
```

Description

`regwrite(id,'regname',value,'represent',timeout)` writes the data in `value` to the `regname` register of the target processor. `regwrite` converts `value` from its representation in the MATLAB® workspace to the representation specified by `represent`. The `represent` input argument defines the format of the data when it is stored in `regname`. Input argument `represent` takes one of three input strings:

represent String	Description
`2scomp`	Write `value` to the destination register as a signed integer value in two's complement format. This is the default setting when you omit the `represent` argument.
`binary`	Write `value` to the destination register as an unsigned binary integer.
`ieee`	Write value to the destination registers as a floating point 32-bit or 64-bit value in IEEE floating-point format. Use this only when you are writing to 32- and 64-bit registers on the target.

String `regname` specifies the name of the destination register on the target. Link `id` defines the target to write `value` to. Valid entries for `regname` depend on your target processor. Register names are not case-sensitive — `a0` is the same as `A0`.

For example, the following registers are some of the many available on the MPC5500 processor:

- 'acc' — Accumulator A register

- `sprg0` through `sprg7` — SPR registers

Other processors provide other register sets. Refer to the documentation for your target processor to determine the registers for the processor.

Note Use `write` (called a direct memory write) to write memory-mapped registers.

To limit the time that `regwrite spends` transferring data to the target processor, the optional argument `timeout` tells the data transfer process to stop after `timeout` seconds. `timeout` is defined as the number of seconds allowed to complete the write operation. You might find this useful for limiting prolonged data transfer operations.

If you omit the `timeout` input argument in the syntax, `regwrite` defaults to the global time-out defined in `id`. If the write operation exceeds the time specified, `regwrite` returns with a time-out error. Generally, time-out errors do not stop the register write process. The write process stops while waiting for MULTI to respond that the write operation is complete.

`regwrite(id,'regname',value,'represent')` omits the `timeout` input argument and does not change the time-out value specified in `id`.

`regwrite(id,'regname',value,)` omits the `represent` input argument. Writing the data does not reformat the data written to `regname`.

Reading and Writing Register Values

Register variables can be difficult to read and write because the registers which hold their value are not dedicated to storing just the variable values.

Registers are used as temporary storage locations at any time during execution. When this temporary storage process occurs, the value of the variable is temporarily stored somewhere on the stack and returned

later. Therefore, getting the values of register variables during program execution may return unexpected answers.

Values that you write to register variables during intermediate times in program operation may not get reflected in the register.

This is true for any local variables as well.

One way to see this is to write a line of code that uses the variable and see if result is consistent.

```
register int a = 100;
int b;
 ...
b = a + 2;
```

Reading the register assigned to `a` may return an incorrect value for `a` but if `b` returns the expected 102 result, nothing is wrong with the code or Embedded IDE Link™ MU.

Examples

To write a new value to the PC register on a C5xxx family processor, enter

```
regwrite(id,'pc',hex2dec('100'),'binary')
```

specifying that you are writing the value 256 (the decimal value of 0x100) to register `pc` as binary data.

To write a 64-bit value to a register pair, such as B1:B0, the following syntax specifies the value as a string, representation, and target registers.

```
regwrite(id,'b1:b0',hex2dec('1010'),'ieee')
```

Registers `B1:B0` now contain the value 4112 in double-precision format.

See Also

`read`, `regread`, `write`

Purpose

Remove file from active project in IDE window

Syntax

```
remove(id,'filename','filetype')
```

Description

`remove(id,'filename','filetype')` removes the file named `filename` from the active project in the `id` window of the IDE. If the file does not exist, MATLAB® returns a warning and does not remove any files. The `filetype` argument is optional, with the default value of `text`. Possible values for `filetype` are: `project` and `text`.

See Also

`add`

`cd`

`open`

reset

Purpose Stop program execution and reset processor

Syntax

```
reset(id,timeout)
```

Description

`reset(id,timeout)` stops the program executing on the processor and asynchronously performs a processor reset, returning all processor register contents to their power-up settings. `reset` returns immediately after the processor halt.

The `timeout` is an optional parameter, with the default value set to the global default value. The `timeout` determines how long, in seconds, MATLAB® waits for the processor to halt.

See Also

`halt`

`load`

`run`

Purpose

Restart in IDE

Syntax

```
restart(id)
restart(id,timeout)
```

Description

`restart(id)` issues a restart command in the MULTI debugger. The behavior of the restart process depends on the processor. Refer to your Green Hills MULTI documentation for details about using restart with various processors.

When `id` is an array that contains more than one processor, each processor calls `restart` in sequence.

`restart(id,timeout)` adds the optional `timeout` input argument. `timeout` defines an upper limit in seconds on the period the restart routine waits for completion of the restart process. If the time-out period is exceeded, `restart` returns control to MATLAB® with a time-out error. In general, `restart` causes the processor to initiate a restart, even if the time-out period expires. The time-out error indicates that the restart confirmation was not received before the time-out period elapsed.

See Also

`halt`

`isrunning`

`run`

Purpose

Execute program loaded on processor

Syntax

```
run(id)
run(id,'runopt')
run(...,timeout)
```

Description

`run(id)` runs the program file loaded on the referenced processor, returning immediately after the processor starts running. Program execution starts from the location of program counter (PC). Usually, the PC is positioned at the top of the executable file. However, if you stopped a running program with `halt`, the PC may be anywhere in the program. `run` starts the program from the PC current location.

If `id` references more the one processor, each processors calls `run` in sequence.

`run(id,'runopt')` includes the parameter `runopt` that defines the action of the `run` method. The options for `runopt` are listed in the following table.

runopt string	Description
`run`	Executes the run and waits to confirm that the processor is running, and then returns to MATLAB®.
`runtohalt`	Executes the run but then waits until the processor halts before returning. The halt can be the result of the PC reaching a breakpoint, or by direct interaction with Green Hills MULTI®, or by the normal program exit process.

`run(...,timeout)` adds input argument `timeout`, to allow you to set the time out to a value different from the global timeout value. The `timeout` value specifies how long, in seconds, MATLAB® waits for the processor to start executing the loaded program before returning.

Most often, the `run` and `runtohalt` options cause the processor to initiate execution, even when a timeout is reached. The timeout

indicates that the confirmation was not received before the timeout period elapsed.

See Also

`halt`

`load`

`reset`

Purpose Set active configuration build options

Syntax

```
setbuildopt(id,tool,ostr)
setbuildopt(id,file,ostr)
```

Description

`setbuildopt(id,tool,ostr)` configures the build options to match the passed OSTR on the specified build `tool`. This replaces the switch settings that are applied when you invoke the command line `tool`. For example, a build tool could be a compiler, linker or assembler. To be sure the `tool` name is defined correctly, use the `getbuildopt` command to read a list of defined build tools. If MULTI® does not recognize OSTR, `setbuildopt` sets all switch settings to default values for the build tool specified by `tool`.

`setbuildopt(id,file,ostr)` configures the build options to match the passed OSTR on the specified source file `file`. The source file must exist in the active project.

See Also

`activate`

`getbuildopt`

Purpose

Visibility of IDE window

Syntax

```
visible(id, state)
```

Description

`visible(id, state)` sets the visibility state of the IDE window defined by `id`. Possible values of `state` are `0` for not visible, and `1` for visible. Setting the state to `1` forces the IDE to be visible on the desktop so you can interact with it. Setting to `0` hides the IDE—the IDE runs in the background. In the not visible state, you interact with the IDE from the MATLAB® command line. When you create a `ghsmulti` object, the IDE visibility is set to `0` and the IDE is not visible.

See Also

`info`

`isvisible`

write

Purpose

Write data to processor memory block

Syntax

```
mem = write(id,address,data)
mem = write(...,datatype)
mem = write(...,memorytype)
mem = write(...,timeout)
```

Description

`mem = write(id,address,data)` writes `data`, a collection of values, to the memory space of the DSP processor referenced by `id`. Input argument `data` is a scalar, vector or array of values to write to the memory of the processor. The block to write begins from the DSP memory location given by the input parameter `address`.

The data is written starting from `address` without regard to type-alignment boundaries in the DSP. Conversely, the byte ordering of the data type is automatically applied.

Note You cannot write data to processor memory while the processor is running.

`address` is a decimal or hexadecimal representation of a memory address in the processor. In all cases, the full memory address consist of two parts: the start address and the memory type. The memory type value can be explicitly defined using a numeric vector representation of the address (see below).

Alternatively, the `id` object has a default memory type value which is applied if the memory type value is not explicitly incorporated into the passed address parameter. In DSP processors with only a single memory type, by setting the `id` object memory type value to zero it is possible to specify all addresses using the abbreviated (implied memory type) format.

You provide the address parameter either as a numerical value that is a decimal representation of the DSP memory address, or as a string that `write` converts to the decimal representation of the start address.

(Refer to function `hex2dec` in the *MATLAB® Function Reference* that `read` uses to convert the hexadecimal string to a decimal value).

To demonstrate how `write` uses `address`, here are some examples of the `address` parameter:

address Parameter Value	Description
`131082`	Decimal address specification. The memory start address is 131082 and memory type is 0. This is the same as specifying [131082 0].
`[131082 1]`	Decimal address specification. The memory start address is 131082 and memory type is 1.
`'2000A'`	Hexadecimal address specification provided as a string entry. The memory start address is 131082 (converted to the decimal equivalent) and memory type is 0.

It is possible to specify `address` as cell array, in which case you can use a combination of numbers and strings for the start address and memory type values. For example, the following are valid addresses from cell array `myaddress`:

```
myaddress1 myaddress1{1} = 131072; myadddress1{2} =
'Program(PM) Memory';
```

```
myaddress2 myaddress2{1} = '20000'; myadddress2{2} =
'Program(PM) Memory';
```

```
myaddress3 myaddress3{1} = 131072; myaddress3{2} = 0;
```

`mem = write(...,datatype)` where the input argument `datatype` defines the interpretation of the raw values written to DSP memory. Parameter `datatype` specifies the data format of the raw memory image. The data is written starting from `address` without regard to data type alignment boundaries in the DSP. The byte ordering of the data type is automatically applied. The following MATLAB® data types are supported:

MATLAB® Data Type	Description
double	IEEE double-precision floating point value
single	IEEE single-precision floating point value
uint8	8-bit unsigned binary integer value
uint16	16-bit unsigned binary integer value
uint32	32-bit unsigned binary integer value
int8	8-bit signed two's complement integer value
int16	16-bit signed two's complement integer value
int32	32-bit signed two's complement integer value

`write` does not coerce data type alignment. Some combinations of `address` and `datatype` will be difficult for the processor to use.

`mem = write(...,memorytype)` adds an optional input argument `memorytype`. Object `id` has a default memory type value 0 that `write` applies if the memory type value is not explicitly incorporated into the passed address parameter. In processors with only a single memory type, it is possible to specify all addresses using the implied memory type format by setting the `id` `memorytype` property value to zero.

Blackfin and SHARC use different memory types. Blackfin processors have one memory type. SHARC processors provide five types. The following table shows the memory types for both processor families.

String Entry for memorytype	Numerical Entry for memorytype	Processor Support
`'program(pm) memory'`	0	Blackfin and SHARC
`'data(dm) memory'`	1	SHARC
`'data(dm) short word memory'`	2	SHARC
`'external data(dm) byte memory'`	3	SHARC
`'boot(prom) memory'`	4	SHARC

`mem = write(...,timeout)` adds the optional parameter *`timeout`* that defines how long, in seconds, MATLAB® waits for the specified write process to complete. If the `timeout` period expires before the write process returns a completion message, MATLAB® throws an error and returns. Usually the process works correctly in spite of the error message.

Examples

These three syntax examples demonstrate how to use `write` in some common ways. In the first example, write an array of 16–bit integers to location [131072 1].

```
write(id,[131072 1],int16([1:100]));
```

Now write a single-precision IEEE floating point value (32-bits) at address 2000A(Hex).

```
write(id,'2000A',single(23.5));
```

For the third example, write a 2-D array of integers in row-major format (standard C programming format) at address 131072 (decimal).

```
mlarr = int32([1:10;101:110]);
```

```
write(id,131072,mlarr');
```

See Also

`hex2dec` in the *MATLAB® Function Reference*

`read`

Blocks — By Category

Blackfin Support (p. 7-2)	Work with Analog Devices® Blackfin® processors
Core Support (p. 7-3)	Work with all supported processors
MPC5500 Support (p. 7-4)	Work with Freescale™ processors
Target Preferences (p. 7-5)	Set target preferences for all supported processors

Blackfin Support

Hardware Interrupt	Generate Interrupt Service Routine
Target Preferences	Configure model for supported-processors

Core Support

Idle Task	Create free-running task
Memory Allocate	Allocate memory section on supported processors
Memory Copy	Copy to and from memory section

MPC5500 Support

HW/SW Interrupt	Generate Interrupt Service Routine
Target Preferences	Configure model for supported-processors

Target Preferences

Target Preferences	Configure model for supported-processors

Blocks — Alphabetical List

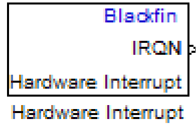
Hardware Interrupt

Purpose

Generate Interrupt Service Routine

Library

Blackfin DSP Support in Embedded IDE Link™ MU



Description

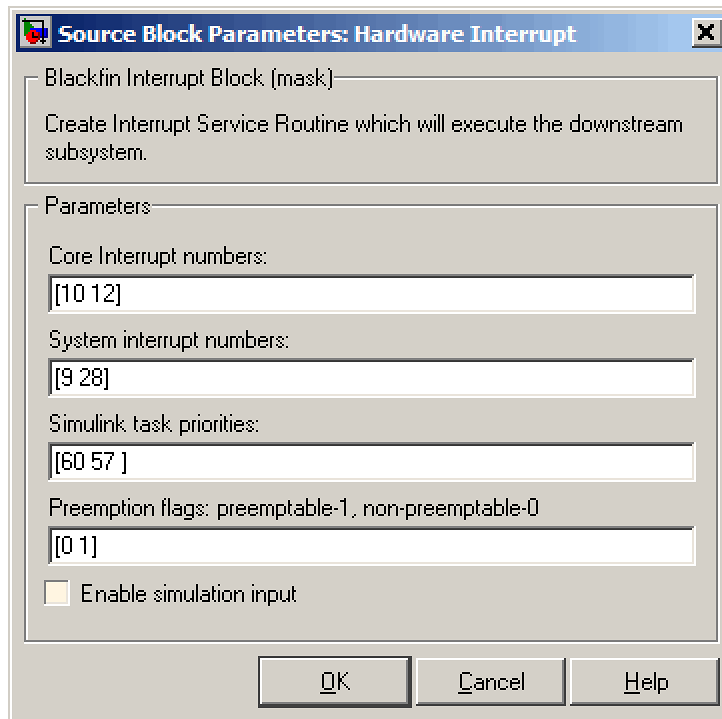
Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from this block or an Idle Task block connected to this block. Core interrupts trigger the ISRs. System interrupts trigger the core interrupts. In the following figure, you see the mapping possibilities between system interrupts and core interrupts.

Interrupts

Blackfin processors support the interrupt numbers shown in the following table. Some Blackfin processors do not support all of the system interrupts.

Interrupt Description	Valid Range in Parameter
Core interrupt numbers	7 to 15
System interrupt numbers	0 to 31 (The upper end value depends on the processor. May be less than 31.)

Dialog Box



Core interrupt numbers

Specify a vector of one or more interrupt numbers for the interrupt service routines (ISR) to install. The valid range is 7 to 15, where 7 through 13 are hardware driven, and 14 and 15 are software driven. Core interrupts numbered 0 to 6 are reserved and cannot be entered in this field.

The width of the block output signal corresponds to the number of interrupt values you specify in this field. Triggering of each ISR depends on the core interrupt value, the system interrupt value, and the preemption flag you enter for each interrupt. These three values define how the code and processor respond to interrupts during asynchronous scheduler operations.

System interrupt numbers

System interrupt numbers identify system interrupts to map to core interrupts. Enter one or more values as a vector. The valid range is 0 through 31, although the valid range depends on your processor. Some processors do not support the full range of 32 system interrupts. The software does not test for valid system interrupt values. You must verify that your values are valid for your processor. You must specify at least one system interrupt number to use asynchronous scheduling.

The block maps the first interrupt value in this field to the first core interrupt value you enter in **Core interrupt numbers**, it maps the second system interrupt value to the second core interrupt value, and so on until it has mapped all of the system interrupt values to core interrupt values. You cannot map more than one system interrupt to the same core interrupt. Therefore, you can enter one system interrupt value in this field and map it to more than one core interrupt. You cannot enter more than one value in this field and map the values to one core interrupt.

When you trigger one of the system interrupts in this field, the block triggers the ISR associated with the core interrupt that is mapped to the system interrupt.

Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink® task priority specifies the Simulink® priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Proper code generation requires rate transition code (see Rate Transitions and Asynchronous Blocks). The task priority values ensure absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

Preemption flags: preemptable – 1, non-preemptable – 0

Higher priority interrupts can preempt interrupts that have lower priority. To control this preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering `1` indicates the corresponding core interrupt can be preempted.
- Entering `0` indicates the corresponding interrupt cannot be preempted.

When **Core interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of preemption flag values that correspond to the order of the interrupts in **Core interrupt numbers**. If **Core interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

For example, the default settings `[0 1]`indicate that the interrupt with value `10` in **Core interrupt numbers** is not preemptible and the value `12` interrupt can be preempted.

Enable simulation input

When you select this option, Simulink® adds an input port to the Hardware Interrupt block. This port receives input only during simulation. Connect one or more simulated interrupt sources to the simulation input.

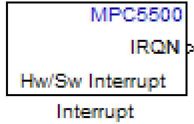
HW/SW Interrupt

Purpose

Generate Interrupt Service Routine

Library

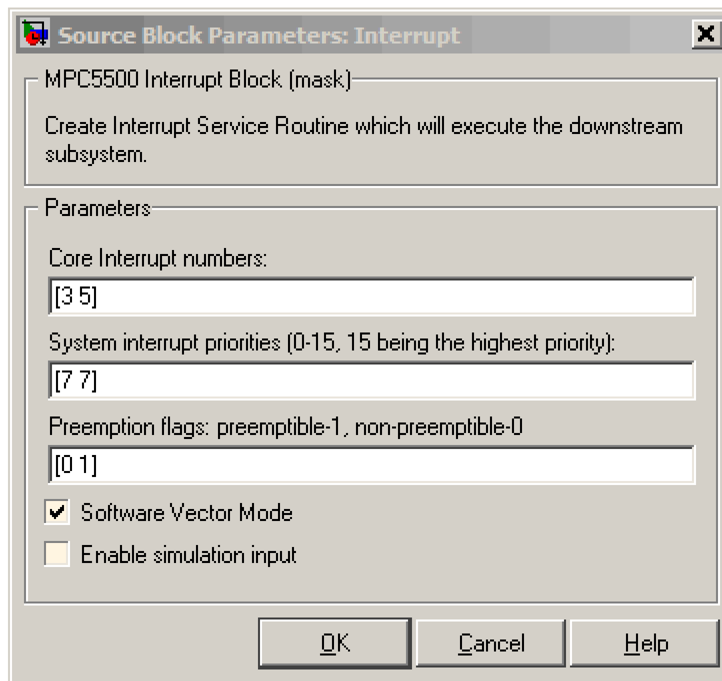
MPC5500 Support in Embedded IDE Link™ MU



Description

Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that either run the processes that are downstream from this block or trigger an Idle Task block connected to this block. Core interrupts trigger the ISRs. System interrupts trigger the core interrupts.

Dialog Box



Core interrupt numbers

Specify a vector of interrupt numbers for the interrupts to install. The block services these interrupts. When your model or code raises one of these interrupts, either through hardware or software, this block reacts to the interrupt and runs the associated downstream block or function. The valid range or interrupts depends on the processor. For example, MPC5553 processors support 212 interrupts. MPC5554 processors support 308 interrupts. Each interrupt in the row vector must be unique. Interrupts that you do not specify in this parameter cause system failures if your project raises them.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. The values in this

field and the preemption flag entries in **Preemption flags: preemptable-1, non-preemptable-0** define how the code and processor handle interrupts during asynchronous scheduler operations.

System interrupt priorities (0–15, 15 being the highest priority)

Each output of the HW/SW Interrupt block drives a downstream block (for example, a function call subsystem). Simulink® task priority specifies the Simulink® priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Core interrupt numbers**. In the default settings shown in the figure, interrupts 3 and 5 have the same priority value—7.

Proper code generation requires rate transition code (see Rate Transitions and Asynchronous Blocks). The task priority values ensure absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

If multiple interrupts share the same priority and are asserted simultaneously, the block selects the lowest numbered interrupt first.

Preemption flags: preemptable – 1, non-preemptable – 0

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering 1 indicates that the interrupt can be preempted.
- Entering 0 indicates the interrupt cannot be preempted.

You cannot set a task that has priority higher than the base rate to be preemptable.

When **Interrupt numbers** contains more than one interrupt value, you can assign different preemption flags to each interrupt

by entering a vector of flag values to correspond to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings `[0 1]`, the interrupt with priority `5` in **Interrupt numbers** is not preemptible and the priority `8` interrupt can be preempted.

Software vector mode

Select this option to put the block and processor in software vector mode. Enabling this option creates a common interrupt handler. Clearing this option puts the processor in hardware vector mode. Refer to the MULTI documentation for more information about the modes.

Enable simulation input

When you select this option, Simulink® adds an input port to the HW/SW Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

Purpose

Create free-running task

Library

DSP Core Support in Embedded IDE Link™ MU

Idle Task

Description

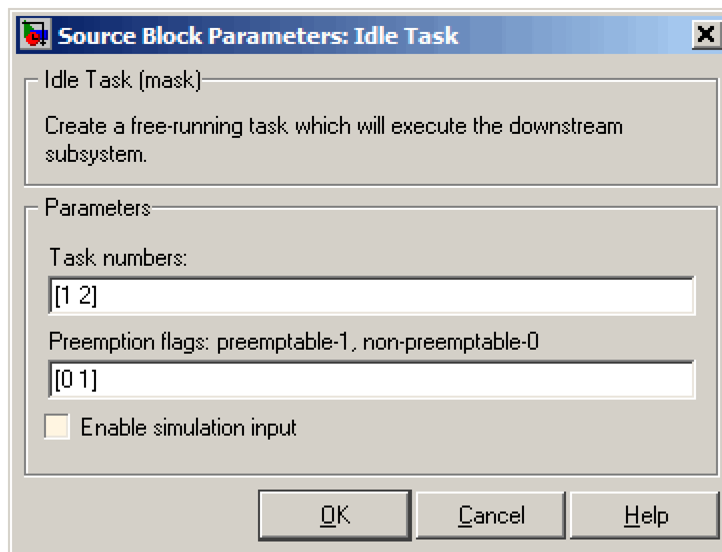
The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. All tasks executed through the Idle Task block are of the lowest priority, lower than that of the base-rate task.

Vectorized Output

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. Any preemption flag vector must be the same length as the number of tasks vector unless the preemption flag vector has only one element. The value of the preemption flag determines whether a given interrupt (and corresponding task) is preemptible. Preemption overrides prioritization. A lower-priority, nonpreemptible task can preempt a higher-priority, preemptible task.

When the preemption flag vector has only one element, that element value applies to all functions in the downstream subsystem as defined by the task numbers in the task number vector. If the preemption flag vector has the same number of elements as the task number vector, each task defined in the task number vector has a preemption status defined by the value of the corresponding element in the preemption flag vector.

Dialog Box



Task numbers

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers. The default values are `[1,2]`, to indicate that the downstream subsystem has two functions.

The values you enter determine the execution order of the functions in the downstream subsystem, while the number of values you enter corresponds to the number of functions in the downstream subsystem.

Enter a vector containing the same number of elements as the number of functions in the downstream subsystem. This vector can contain no more than 16 elements, and the values must be from 0 to 15 inclusive.

The value of the first element in the vector determines the order in which the first function in the subsystem is executed. Similarly,

the value of the second element determines the order in which the second function in the subsystem is executed, and so on.

For example, entering the vector [2,3,1] in this field indicates that there are three functions to be executed, and that the third function is executed first, the first function is executed second, and the second function is executed third. After all functions are executed, the Idle Task block cycles back and repeats the execution of the functions in the same order.

Preemption flags

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering `1` indicates that the interrupt can be preempted.
- Entering `0` indicates the interrupt cannot be preempted.

When **Task numbers** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of the tasks in **Task numbers**. If **Task numbers** contains more than one task, and you enter only one flag value in this field, that preemption setting applies to all tasks.

For example, the default settings `[0 1]` indicate the task with priority `1` in **Task numbers** is not preemptible, and the priority `2` task can be preempted.

Enable simulation input

When you select this option, Simulink® adds an input port to the Idle Task block. This port receives input only during simulation. Connect one or more simulated interrupt sources to the simulation input.

Note Select this check box to test asynchronous interrupt processing behavior in Simulink®.

Memory Allocate

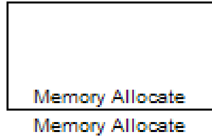
Purpose

Allocate memory section on supported processors

Library

Core Support in Embedded IDE Link MU

Description



Memory Allocate

On your processor, this block directs the Green Hills MULTI® compiler to allocate memory for a new variable you specify. Parameters in the block dialog box let you specify the variable name, the alignment of the variable in memory, the data type of the variable, and other features that fully define the memory required.

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

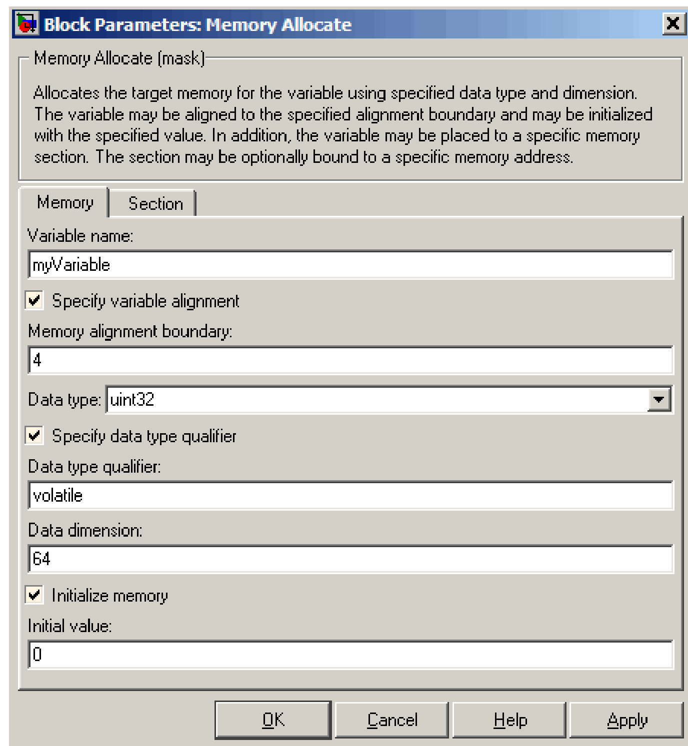
The block does not have input or output ports. It only allocates a memory location. You do not connect it to other blocks in your model.

Dialog Box

The block dialog box comprises multiple tabs:

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

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



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

Memory Parameters

You find the following memory parameters on this tab.

Variable name

Specify the name of the variable to allocate. The variable is allocated in the generated code.

Specify variable alignment

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory alignment boundary** parameter appears so

you can specify the alignment. Use this parameter and **Memory alignment boundary** when your processor requires this feature.

Memory alignment boundary
After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

Data type
Defines the data type for the variable. Select from the list of types available.

Specify data type qualifier
Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

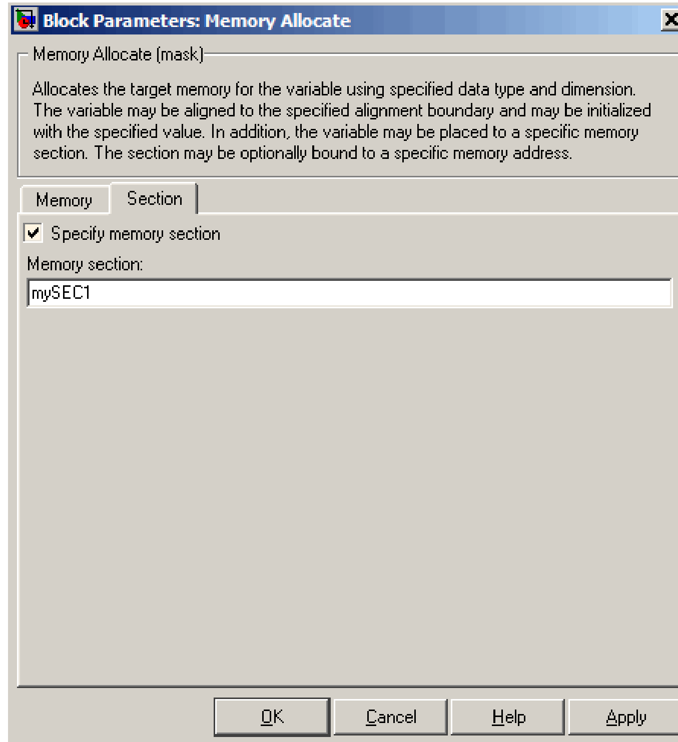
Data type qualifier
After you select **Specify data type qualifier**, you enter the desired qualifier here. `Volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

Data dimension
Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

Initialize memory
Directs the block to initialize the memory location to a fixed value before processing.

Initial value
Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.

Section Parameters



Parameters on this pane specify the section in memory to store the variable.

Specify memory section

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the standard memory sections or a custom section that you declare elsewhere in your code.

Memory section

Identify a specific memory section to allocate the variable in **Variable name**. Verify that the section has sufficient space to store your variable. After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use **Bind memory section** to bind the memory section to a location.

See Also

Memory Copy

Memory Copy

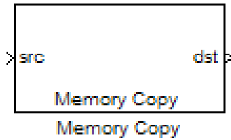
Purpose

Copy to and from memory section

Library

Core Support in Embedded IDE Link™ MU

Description



Memory Copy

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

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

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

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

Block Operations

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in all three periods independently.

Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. This block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform any operation. The block output is not defined.

Copy Memory

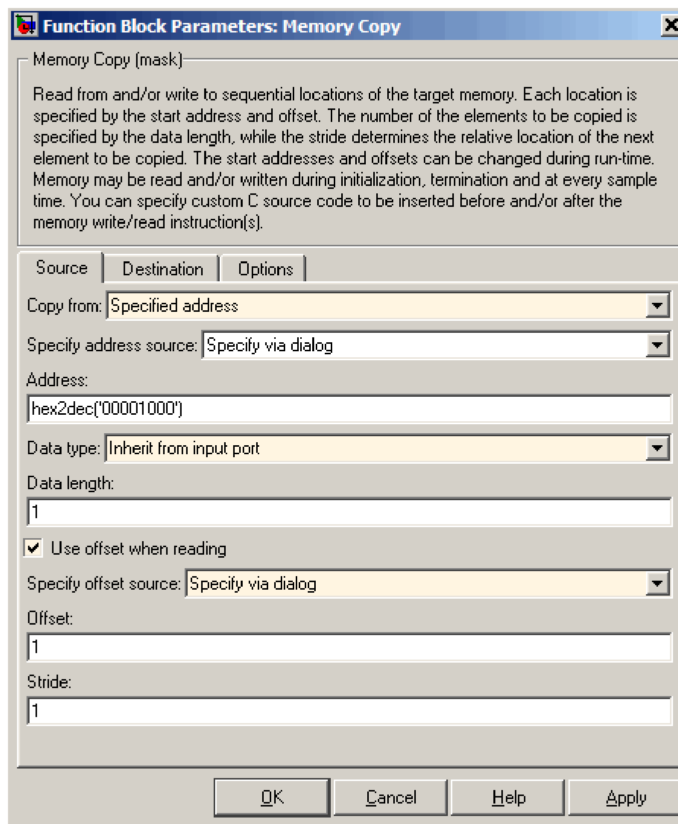
When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

Dialog Box

The block dialog box contains multiple tabs:

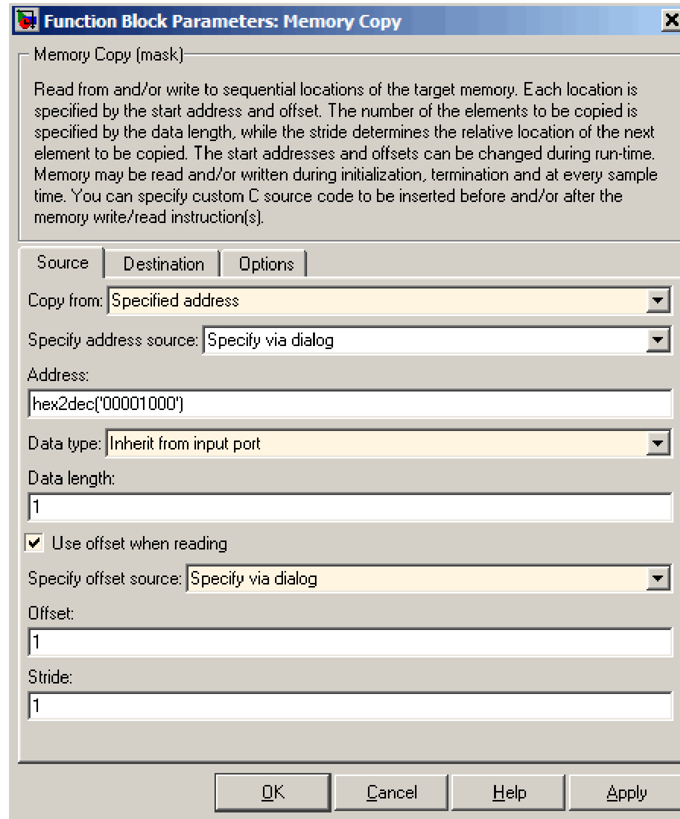
- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

The dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.



Sections that follow describe the parameters on each tab in the dialog box.

Source Parameters



Copy from

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

- `Input port` — This source reads the data from the block input port.
- `Specified address` — This source reads the data at the specified location in **Specify address source** and **Address**.

- `Specified source code symbol` — This source tells the block to read the symbol (variable) you enter in **Source code symbol**. When you select this copy from option, you enable the **Source code symbol** parameter.

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

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

Specify address source

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

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

Source code symbol

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

Address

When you select `Specify via dialog` for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use `hex2dec` to convert the address to the proper format. The following example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either `4096` or `hex2dec('1000')` as the address.

Data type

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

Data length

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

Use offset when reading

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

Specify offset source

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

Offset

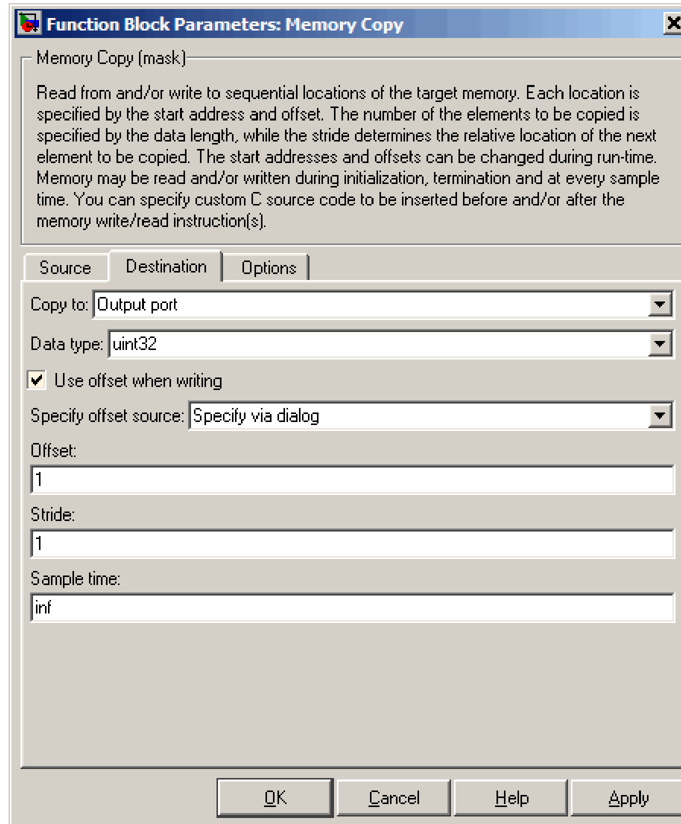
Offset tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

Stride

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in the source address equal to the stride. **Stride** must be a positive integer.

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

Destination Parameters



Copy to

Select the destination for the data. Choose one of the entries on the list:

- `Output port` — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.

- `Specified address` — Copies the data to the specified location in **Specify address source** and **Address**.
- `Specified source code symbol` — Tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.

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

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

Specify address source

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

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

Source code symbol

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this

symbol. The block does not verify that the symbol exists and uses valid syntax.

Address

When you select `Specify via dialog` for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use `hex2dec` to convert the address to the proper format. This example converts `0x2000` to decimal form.

```
8192 = hex2dec('2000');
```

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

Data type

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

Specify offset source

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

Offset

Offset tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before

writing the first value to the destination. One is the default offset value and **Offset** accepts only positive integers of one or greater.

Stride

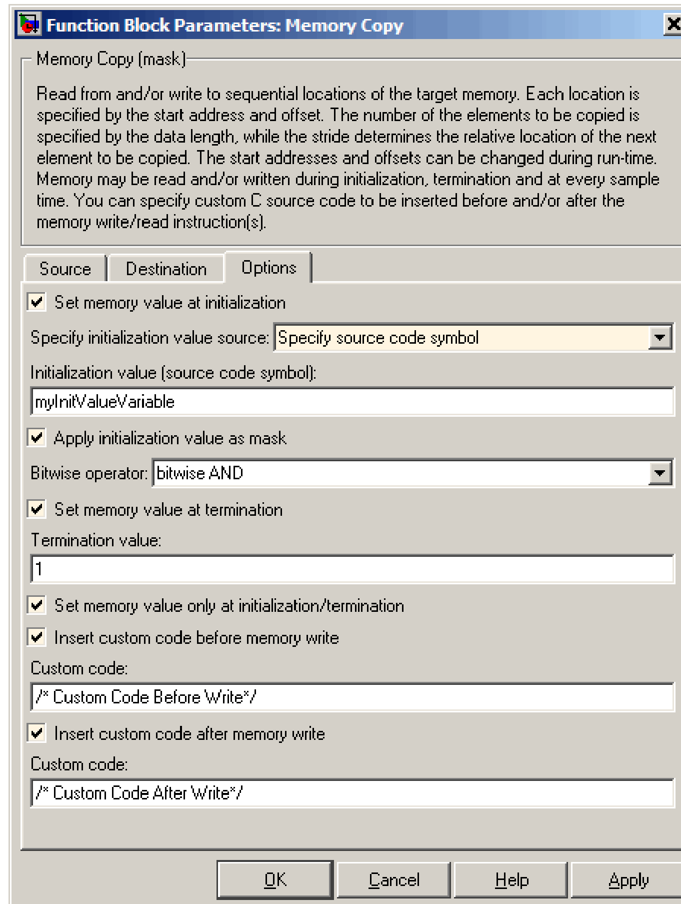
Stride lets you specify the spacing for copying the input to the destination. By default, the stride value is one, meaning the generated code writes the input data sequentially to the destination in consecutive locations. When you add a stride value not equal to one, the output data is stored not sequentially,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

This figure shows a stride value of three applied to writing the input to an output location. You can specify a stride value for the input with parameter **Stride** on the **Source** pane. As shown in the figure, you can use both an input stride and output stride at the same time to enable you to manipulate your memory more fully.

Sample time

Sample time sets the rate at which the memory copy operation occurs, in seconds. The default value `Inf` tells the block to use a constant sample time. You can set **Sample time** to `-1` to direct the block to inherit the sample time from the input, if there is one, or the Simulink® model (when there are no input ports on the block). Enter the sample time in seconds as you need.

Options Parameters



Set memory value at initialization

When you check this option, you direct the block to initialize the memory location to a specific value when you initialize your program at run time. After you select this option, use the **Set memory value at termination** and **Specify initialization**

value source parameters to set your desired value. Alternately, you can tell the block to get the initial value from the block input.

Specify initialization value source

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

- `Specify constant value` — Sets a single value to use when your program initializes memory. Enter any value that meets your needs.
- `Specify source code symbol` — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a string.

Initialization value (constant)

If you check **Set memory value at initialization** and choose `Specify constant value` for **Specify initialization value source,** enter the constant value to use in this field. Any real value that meets your needs is acceptable.

Initialization value (source code symbol)

If you check **Set memory value at initialization** and choose `Specify source code symbol` for **Specify initialization value source,** enter the symbol to use in this field. Any symbol that meets your needs and is in the symbol table for the program is acceptable. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

Apply initialization value as mask

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol.

Bitwise operator

To use the initialization value as a mask, select one of the entries on the following table from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

Bitwise Operator List Entry	Description
`bitwise AND`	Apply the mask value as a bitwise AND to the value in the register.
`bitwise OR`	Apply the mask value as a bitwise OR to the value in the register.
`bitwise exclusive OR`	Apply the mask value as a bitwise exclusive OR to the value in the register.
`left shift`	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
`right shift`	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

Set memory value at termination

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

Set memory value only at initialization/termination

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform any copies during real-time operations.

Insert custom code before memory write

Select this parameter to add custom C code before the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your C code.

Custom code

Enter the custom C code to insert into the generated code just before the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

Insert custom code after memory write

Select this parameter to add custom C code immediately after the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your C code.

Custom code

Enter the custom C code to insert into the generated code just after the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

See Also

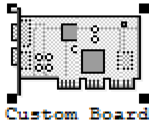
Memory Allocate

Purpose

Configure model for supported-processors

Library

Target Preferences in Embedded IDE Link™ MU



Description

Options on the block mask let you set features of code generation for your custom Blackfin, NEC V850, MPC5500, or MPC7400 processor. Adding this block to your Simulink® model provides access to the processor hardware settings you need to configure when you generate code from Real-Time Workshop® to run on the processor.

Any model that you target to custom hardware must include this block. Real-Time Workshop® returns an error message if a target preferences block is not present in your model.

Note This block must be in your model at the top level and not in a subsystem. It does not connect to other blocks, but stands alone to set the target preferences for the model. Simulink® returns an error when your model either does not include a target preferences block or has more than one.

You can specify the following processor and target options on this block:

- Board and processor information
- Memory mapping and layout
- Allocation of the various code sections, such as compiler, and custom sections

Setting the options included in this dialog box results in identifying your target to Real-Time Workshop®, MULTI, and Simulink®, and

configuring the memory map for your target. Both steps are essential for developing code for any processor that is custom or explicitly supported.

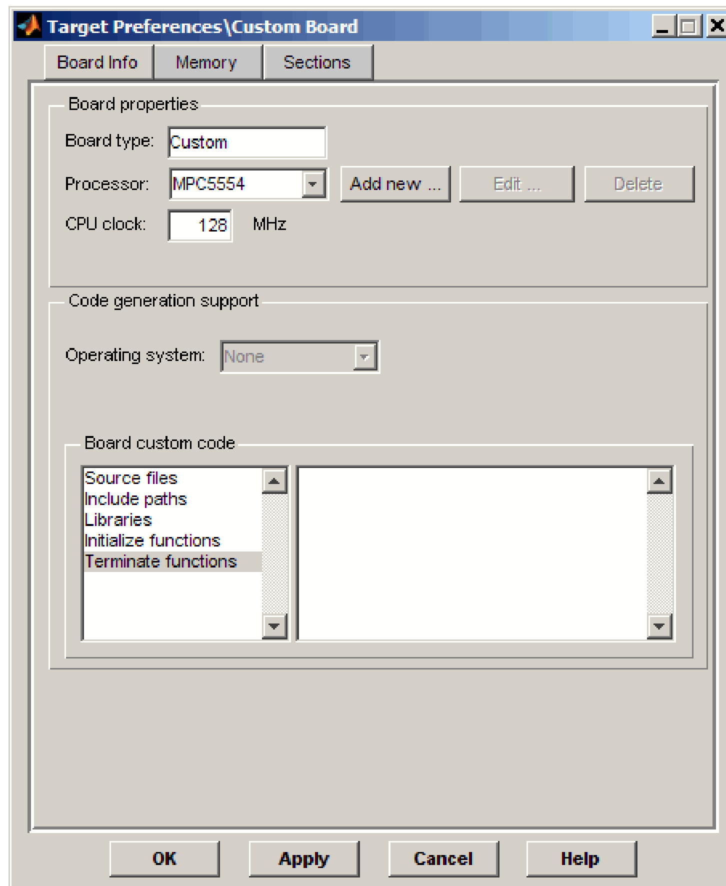
Unlike most other blocks, you cannot open the block dialog box for this block until you add the block to a model. When you try to open the block dialog box, the block attempts to connect to a Green Hills MULTI® session. It cannot make the connection when the block is in the library. If you try to open the block dialog box before you add it to a model, the open process fails and returns an error message.

Note If you do not have Green Hills MULTI® installed on your PC, you cannot open this block dialog box.

Generating Code from Model Subsystems

Real-Time Workshop® provides the ability to generate code from a selected subsystem in a model. To generate code for a supported processor from a subsystem, the subsystem model must include this target preferences block.

Dialog Box



All target preferences block dialog boxes provide tabbed access to the following panes. You set the options for the processor from these panes:

- **Board info** — Select the board type and processor, set the clock speed, and identify the session.
- **Memory** — Set the memory allocation and layout on the processor (memory mapping).

- **Sections** — Determine the arrangement and location of the sections on the processor, such as where to put the compiler information.

Board Info Pane

The following options appear on the **Board Info** pane for the **Target Preferences** dialog box.

Board Type
Enter `Custom` for this option.

Processor
Select the processor from the list. The processor type you enter determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box. If you select NEC V850 from the list, the Memory and Sections tabs do not appear on the dialog box. You cannot change the memory map and section allocation for the NEC V850.

CPU clock
Shows the clock speed of the processor. When you enter a value, you are not changing the CPU clock rate. Instead, you are reporting the actual rate. If the value you enter does not match the rate on the processor, your model's real-time results code profiling results may be incorrect.

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting **CPU clock** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

When you generate code for targets from Simulink® models, you may encounter the software timer. The timer is invoked automatically to handle and create interrupts to drive your model if the processing rates in your model change (the model is multirate).

Correctly generating interrupts for your model depends on the clock rate of the CPU on your processor.

For the timer software to calculate the interrupts correctly, MULTI needs to know the actual clock rate of your processor as you configured it. CPU clock speed lets you tell the timer the rate at which your processor CPU runs, which is the rate to use to match the CPU rate.

The timer uses the CPU clock rate you specify in **CPU clock** to calculate the time for each interrupt. For example, if your model includes a sine wave generator block running at 1 kHz feeding a signal into an FIR filter block, the timer needs to create interrupts to generate the sine wave samples at the proper rate. The timer uses the clock rage you enter, for example, 100 MHz, to calculate the sine generator interrupt period as follows for the sine block:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

100,000,000/1000 = 1 Sine block interrupt per 100,000 clock ticks

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

Board Custom Code

Entries in this group let you specify the locations of custom source files or libraries or other functions. Five options provide access to text areas where you enter files and file paths:

- `Source files` — Enter the full paths to source code files to use with this processor. By default, there are no entries in this parameter.
- `Include paths` — If you require additional files on your path, add them by typing the path into the text area. The default setting does not include additional paths.
- `Libraries` — These entries identify specific libraries that the processor requires. They appear on the list by default if

required. Add more as you require by entering the full path to the library with the library file in the text area. No additional libraries appear in this field in the default configuration.

- `Initialize functions` — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.
- `Terminate functions` — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

When you enter a path to a file, library, or other custom code, use the string

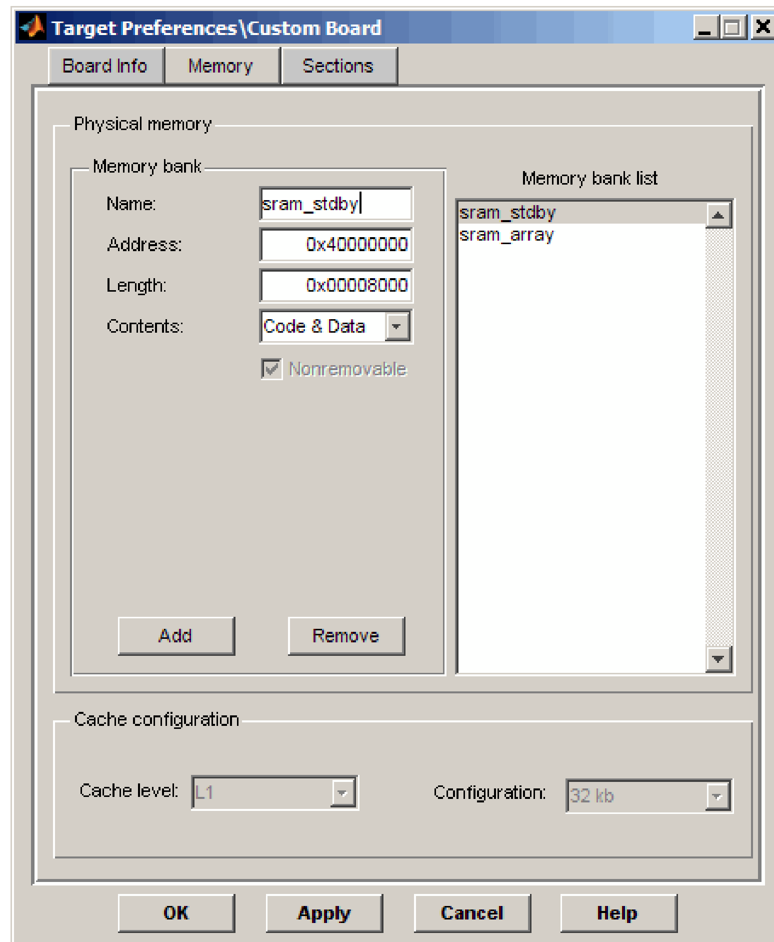
```
$(install_dir)
```

to refer to the MULTI installation directory.

Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code. **Board custom code** options do not support functions that use return arguments or values. Only functions of type `void fname void` are valid as entries in these parameters.

Memory Pane

When you develop models for any processor, you need to specify the layout of the physical memory on your processor and board to determine how use it for your program. For supported boards, the board-specific target preferences blocks set the default memory map. This pane does not apply to NEC V850 processors or Freescale MPC7400 processors..



The **Memory** pane contains memory options in three areas:

- **Physical Memory** — Specify the processor and board memory map
- **Heap** — Specify whether you use a heap and determine the size in words

- **Cache configuration** — Enables the cache (where available) and sets the size in kilobytes

---

Note Your **Physical Memory**, **Heap**, and **Cache configuration** settings on this pane may affect options on the **Sections** pane. Your choices on the **Memory** pane may change how you configure some options on the **Sections** pane.

Most of the information about memory segments and memory allocation is available from the online help system for MULTI.

Physical Memory Options

This list shows the physical memory segments available on the board and processor. By default, target preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured target preferences blocks shows memory segments that are available on the board, but that are external to the processor (external memory). Target preferences blocks set default starting addresses, lengths, and contents of the default memory segments.

The default memory segments for each processor and board are different. For example:

- Custom boards based on Blackfin processors provide `L1_Scratch_SRAM` memory segments by default.
- MPC5500 boards provide `sram_stdby` memory segments by default.

Name

When you highlight an entry on the **Physical memory** list, the name of the entry appears in this field. To change the name of the existing memory segment, select it in the **Physical memory** list and then type the new name in this field.

Note You cannot change the names of default processor memory segments.

To add a new physical memory segment to the list, click **Add**, replace the temporary label in **Name** with the one to use, and press **Return**. Your new segment appears on the list.

After you add the segment, you can configure the starting address, length, and contents for the new segment. New segments start with code and data as the type of content that can be stored in the segment (refer to the **Contents** option).

Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

Address

Address reports the starting address for the memory segment showing in the **Name** field. Address entries are in hexadecimal format and limited only by the board or processor memory.

When you are using a processor-specific preferences block, the starting address shown is the default value. You can change the starting value by entering the new value directly in the **Address** field when you select the memory segment to change.

Length

From the starting address, **Length** sets the length of the memory allocated to the segment in the **Name** field. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs).

When you are using a processor-specific preferences block, the length shown is the default value. You can change the value by entering the new value directly in this option.

Contents

Contents details the kind of program sections that you can store in the memory segment in **Name**. As the processor type for the target preferences block changes, the kinds of information you store in listed memory segments may change. Generally, the **Contents** list contains these strings:

- `Code` — Allow code to be stored in the memory segment in **Name**.
- `Data` — Allow data to be stored in the memory segment in **Name**.
- `Code and Data` — Allow code and data to be stored in the memory segment in the **Name** field. When you add a new memory segment, this string is the default setting for the contents of the new element.

You may add or use as many segments of each type as you need, within the limits of the memory on your processor.

Add

Click **Add** to add a new memory segment to the processor memory map. When you click **Add**, a new segment name appears, for example `NEWMEM1`, in **Name** and on the **Physical memory** list. In **Name**, change the temporary name `NEWMEM1` by entering the new segment name. Entering the new name or clicking **Apply** updates the temporary name on the list to the name you enter in this field.

Remove

Remove a memory segment from the memory map. Select the segment to remove on the **Physical memory** list, and click **Remove** to delete the segment.

Memory bank list

Displays all available memory banks for the selected processor. When you select one of the entries on this list, the **Name**, **Address**, **Length**, and **Contents** parameters change to reflect the memory block selection. With the contents list, you can change the type of material stored in the block—data or code or both.

Create Heap (where applicable)

If your processor supports using a heap, as the Blackfin processors do, selecting this option enables creating the heap, and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

Use this option to create a heap in any memory segment on the **Physical memory** list. Select the memory segment on the list, and then select **Create heap** to create a heap in the selected segment. After you create the heap, use the **Heap size** and **Define label** options to configure the heap.

The location of the heap in the memory segment is not under your control. The only way to control the location of the heap in a segment is to make the segment and the heap the same size. Otherwise, the compiler determines the location of the heap in the segment.

Heap Size (where applicable)

After you select **Create heap**, this option lets you specify the size of the heap in words. Enter the number of words in decimal format. When you enter the heap size in decimal words, the system converts the decimal value to hexadecimal format. You can enter the value directly in hexadecimal format as well. Processors may support different maximum heap sizes.

Define Label (where applicable)

Selecting **Create heap** enables this option that allows you to name the heap. Enter your label for the heap in the **Heap label** option.

Heap Label (where applicable)

Use this option which you enable by selecting **Define label**, to provide the label for the heap. Any combination of characters is accepted for the label, except reserved characters in C/C++ compilers.

Cache level

Blackfin processors support different cache arrangements. For Blackfin processors, you can select one of the following options from the list:

- L1_Code_CACHE
- L1_DataA_CACHE
- L1_DataB_CACHE

Freescale processors do not support cache storage.

Configuration (where applicable)

Select the size of the cache from the list to determine the size of the cache allocated. Blackfin processors support 0 bits or 16 bits.

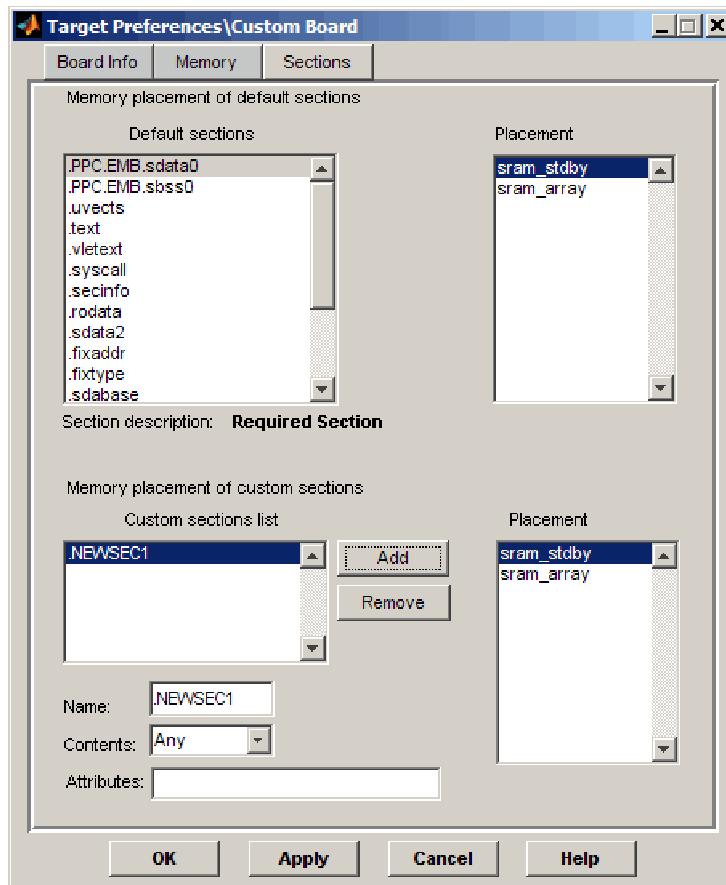
Sections Pane

Options on this pane let you specify where various program sections should go in memory. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations. Commonly used sections include `.program`, `.bsz`, `.data1`, and `.stack`. This pane does not apply for NEC V850 or Freescale MPC7400 processors.

As you change the processor you select on the Board Info pane, the options and list entries on this pane change.

For more information about program sections and objects, refer to the Green Hills MULTI online help.

The following figure shows the Memory pane as it appears for the Blackfin processors.



Within this pane, you allocate the memory needed for the **Default sections** and **Custom sections**.

You can learn more about memory sections and objects in your Green Hills Software MULTI online help.

Default sections

During program compilation, the compiler produces both uninitialized and initialized blocks of data and code. These blocks

get allocated into memory as required by the configuration of your system. On the **Default sections** list you find both initialized sections that contain data or executable code and uninitialized sections that reserve space in memory.

For Blackfin processors, the initialized sections are:

- `bsz`
- `bsz_init`
- `constdata`
- `seg_pmco`
- `seg_pmda`
- `voldata` (created by the assembler)

These sections are uninitialized:

- (created by the assembler)
- `heap`
- `stack`
-

Other sections appear on the list as well:

- (created by the assembler)
-
-

Note The C/C++ compiler does not use the `.data` section.

When you highlight a section on the list, **Section description** shows a brief description of the section. Also, **Placement** shows you where the section is presently allocated in memory.

Section description
: Describes the contents of the selected entry on the **Default sections** list.

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

Custom sections
: When your program uses code or data sections that are not included in the **Default sections** list, you add the new sections to this list. Initially, the **Custom sections** list contains no fixed entries, but instead, only a placeholder for a section for you to define.

Name
: Enter the name for your new section in this field. To add a new section, click **Add**. Then, replace the temporary name with the name to use. Although the temporary name includes a period at the beginning you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

Placement
: After you have added the new section to the **Name** list, select the memory segment to which to add your new section. Limited only by the restrictions imposed by the hardware and compiler, you can select any segment that appears on the list.

Add

Click **Add** to add a new entry on the list of custom sections. When you click **Add**, a new temporary name, for example `NEWMEM1` appears in the **Name** field. Enter the new custom section name to add the section to the **Custom sections** list. After you enter the new name, click **Apply** to add the new section to the list. You can also click **OK** to add the section to the list and close the dialog box.

Remove

Remove a section from the **Custom sections** list. To remove a section, select the section, and click **Remove**.

New Processor Dialog Box

Clicking **Add new** on the **General pane**opens this dialog box to add a new processor to the list of supported processors.

The first time you click **Save** to add a new processor definition to the list of supported processors, a dialog box opens that directs you to select a destination folder for the saved processor definitions file `customChipInfo.dat`. You must select a directory to which you have write access. The location you specify becomes part of your MATLAB® preferences. Future processors that you add become entries in the file `customChipInfo.dat`.

To add a new processor, you must enter values for the following parameters:

- **Name**
- **Processor Class**
- **CPU clock**
- **Compiler switch**
- **Code generation hook**
- **Define internal memory banks** (one or more memory banks)
- **Define default sections** (one or more default sections)

If you do not provide an entry for each of these parameters, Embedded IDE Link™ MU returns an error message and does not create the new processor entry.

General

Name

Provide a name to identify your new processor. You can use any valid C string in this field. The name you enter appears on the list of processors after you add the new processor.

CPU clock

Enter the clock speed of the processor in MHz. When you enter a value, you are not setting the CPU clock rate on the processor. You are reporting the rate. If the value you enter does not match the rate on the processor, your model's real-time results and code profiling results may not be correct.

Setting **CPU clock** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.

Processor class

This represents the class for the new processor. New processors must be members of processor families that Embedded IDE Link™ MU supports, such as a new Blackfin processor or a new Freescale MPC processor.

Generally, processors in a family share common design elements such as interrupt architecture and clock. They may have different memory maps. By selecting the processor class, you identify the common features of the processor family. The parameters in **Define internal memory banks** and **Define default sections** enable you to specify the memory mapping for your new processor.

For example, to add a new Blackfin processor, enter the string TBD.

Compiler switch

Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class. For example, to set the compiler switch for a new Blackfin processor, enter TBD.

Code generation hook

This string specifies a prefix to add when the code generation process calls certain hook functions. The hook allows the code to call into handling functions that are specific to the processor selected.

Define internal memory banks

Name

To add a new physical memory segment to the internal memory banks list, click **Add**, replace the temporary label in **Name** with the one to use, and press **Return**. Your new segment appears on the list.

After you add the segment, you can configure the starting address, length, and contents for the new segment. New segments start with code and data as the type of content that can be stored in the segment (refer to the **Contents** option).

Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

Address

Address reports the starting address in hexadecimal format for the memory segment showing in **Name**. Address entries are limited only by the board or processor memory.

When you are using a processor-specific preferences block, the starting address shown is the default value. You can change the starting value by entering the new value directly in **Address** when you select the memory segment to change.

Length

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs). For the Green Hills Software processor families, for example, the MADU is 8 bytes, 1 word.

Contents

Contents specifies the kind of program sections that you can store in the memory segment in **Name**. When you change the processor type, the kinds of information you can store in listed memory segments may change. Generally, the **Contents** list contains these strings:

- `Code` — Allow code to be stored in the memory segment in **Name**.
- `Data` — Allow data to be stored in the memory segment in **Name**.
- `Code and Data` — Allow code and data to be stored in the memory segment in **Name**. When you add a new memory segment, this setting is the default for the contents of the new element.

You may add or use as many segments of each type as you need, within the limits of the memory on your processor.

Add

Click **Add** to add a new memory segment to the processor memory map. When you click **Add**, a new segment name appears, for example `NEWMEM1`, in **Name** and on the list. In **Name**, change the temporary name `NEWMEM1` by entering the new segment name. Entering the new name, or clicking **OK** updates the temporary name on the list to the name you enter.

Remove

Remove a memory segment from the memory map. Select the segment to remove from the list, and click **Remove** to delete the segment.

Define cache configuration

Label

Enter your label for the cache in the **Label** field. Entering the label updates the label of the selected configuration.

Start

Enter the starting address for the new cache configuration. The address should be a hexadecimal value starting with `0x`.

Growth

Select whether the cache grows in the `Positive` direction or in the `Negative` direction from the starting address.

Options

Enter the label for each option of the selected cache configuration, one label on each line, such as 0kb, 16kb, 32kb and so on.

Add

Click **Add** to add a new cache configuration to the list. When you click **Add**, the new cache label appears on the list.

Remove

Remove a cache configuration from the cache list. Select the configuration to remove from the list, and click **Remove** to delete the cache.

Cache configurations and related options are defined as symbols to the project generator component. Cache options for new processors are not labeled until you add the labels.

Define Default Sections

Options in this region let you specify where various program sections go in memory and the contents and label for each section. You can add text to describe each section. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations. Commonly used sections include `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler, and some can be custom sections as you require.

Label

The name of the section corresponds to the symbolic name recognized by the linker program used with the respective processor.

Description

Enter text that describes the default section to add.

Contents

Contents provides the information about the native of the program section. As the processor type for the target preferences block changes, the kinds of information you store in listed sections may change. Generally, the **Contents** list contains these strings:

- `Code` — Allow code to be stored in the section in **Label**.
- `Data` — Allow data to be stored in the section in **Label**.

You may add or use as many sections of each type as you need, within the limits of the memory on your processor.

Placement

Select the default placement for the new section from the list of available sections.

Add

Click **Add** to add a new section to the list. Clicking **Add** enables the parameters that define the new section.

Remove

This option lets you remove a section from the section list. Select the section to remove from the list, and click **Remove** to delete the section.

Sections and related options are defined as symbols to the project generator component. Section options for new processors are not labeled until you add the labels.

Processor Custom Code

The list on the left side of the pane shows the kinds of custom code you can specify for your processor. Each time you use your custom processor as defined in this dialog box, the custom code you enter in this field applies. You can enter custom code in the categories in the following table.

Custom Code Entry	Description
`Source files`	Enter the full paths to source code files to use with this processor. By default there are no entries in this parameter. Enter each source file on a new line.
`Include paths`	If you require additional header files on your path, add them by typing the path into the text area, one file per line. The default setting does not include additional paths.
`Libraries (Little Endian)`	These entries identify specific little endian libraries that the processor requires. Add more as you require by entering the full path to the library with the library file in the text area. Enter one library per line. No additional libraries appear in the default configuration.
`Libraries (Big Endian)`	These entries identify specific big endian libraries that the processor requires. Add more as you require by entering the full path to the library with the library file in the text area. No additional libraries appear in the default configuration. Enter one library per line.
`Preprocessor symbols`	Enter any preprocessor symbols that the new processor requires for operation and compilation. No preprocessor symbols appear in the default configuration. Add the required symbols one symbol per line.

You can use two types of tokens when you specify custom code paths:

- `$(Install_dir)` — Refers to the installation directory of Green Hills MULTI®. One example of this token is

```
$(Install_dir) \multi\csl\lib\...
```

- `$(MATLAB_ROOT)` — Refers to the directory where you installed MATLAB®.

A

Examples

Use this list to find examples in the documentation.

Automation Interface

"Getting Started with Automation Interface" on page 2-2

Working with Links

"Example — Constructor for ghsmulti Objects" on page 2-20
"Example — Setting Link Property Values at Construction" on page 2-23
"Example — Setting Link Property Values Using set" on page 2-24
"Example — Retrieving Link Property Values Using get" on page 2-24
"Example — Direct Property Referencing in Links" on page 2-24

Asynchronous Scheduler

"Asynchronous Scheduler Examples" on page 3-12
"Idle Task" on page 3-15
"Hardware Interrupt Triggered Task" on page 3-16

Project Generator

"Project Generator Tutorial" on page 3-18

Verification

"PIL Block" on page 4-6
"Real-Time Execution Profiling" on page 4-12

Index

A

access properties 2-22
Archive_library 3-47
asynchronous scheduling 3-9

B

block limitations using model reference 3-48

C

compiler options string, set compiler options 3-41
configure the software timer 8-38
connect to simulator 6-21
CPU clock speed 8-38
Create_project option 3-38
current CPU clock speed 8-38

D

debug operation
    new 6-43
discrete solver 3-27

E

Embedded IDE Link™ MU
    use multilinklib blocks 3-3
Embedded IDE Link™ MU build options
    Create_project 3-38
execution in timer-based models 3-10

F

file and project operation
    new 6-43
fixed-step solver 3-27
functions
    overloading 2-25

G

generate optimized code 3-37
getting properties 2-24
ghsmulti 2-20
ghsmulti object properties 2-27
    portnum 2-27
    procnum 2-27
Green Hills MULTI® IDE objects
    tutorial about using 2-2
Green Hills Software
    general code generation options 3-35
    processor options 3-30
    run-time options 3-37
    TLC debugging options 3-34
Green Hills Software model reference 3-45
Green Hills Software processor
    code generation options 3-37

I

info 6-25
Inline Signal Processing Blockset functions
    option 3-37
issues, using PIL 4-6

L

link filters properties
    getting 2-24
link properties
    about 2-26
    setting 2-24
link properties, details about 2-26
links
    closing Green Hills MULTI® 2-18
    details 2-26
    loading files into Green Hills MULTI®
        IDE 2-11
    quick reference 2-26
    working with your processor 2-13

list 6-32
list object 6-32
list variable 6-32

M

Memory Allocate block 8-14
Memory Copy block 8-20
model execution 3-9
model reference 3-45
- about 3-45
- `Archive_library` 3-47
- block limitations 3-48
- `modelreferencecompliant` flag 3-48
- setting build action 3-47
- target preferences blocks 3-47
- using 3-47

model schedulers 3-9
`modelreferencecompliant` flag 3-48
MULTI
- starting from MATLAB 2-5
- stopping from MATLAB 2-5

O

object
- `ghsmulti` 2-20

object properties
- quick reference table 2-26

objects
- creating objects for Green Hills MULTI® IDE 2-9
- introducing the objects for Green Hills MULTI® IDE tutorial 2-2
- tutorial about using Automation Interface for Green Hills MULTI® IDE 2-2

optimization, processor-specific 3-37
overloading 2-25

P

PIL block 4-6
PIL cosimulation
- definitions 4-4
- how cosimulation works 4-5
- overview 4-3

PIL issues 4-6
portnum 2-27
processor configuration options
- overrun action 3-39

processor information, get 6-25
processor-specific optimization 3-37
`procnum` 2-27
project options
- compiler options string 3-41
- stack size 3-41

properties
- link properties 2-26
- referencing directly 2-24
- retrieving 2-22
  - function for 2-24
- retrieving by direct property referencing 2-24
- setting 2-22

R

read register 6-52
Real-Time Workshop options
- generate code only 3-34

Real-Time Workshop solver options 3-27
regread 6-52
regwrite 6-56
run-time options
- overrun action 3-39

run—time options
- build action 3-38

S

set overrun action, overrun action 3-39

set properties 2-22
set stack size 3-41
simulator
    connect to 6-21
solver option settings 3-27
stack size, set stack size 3-41
start MULTI from MATLAB 2-5
stop MULTI from MATLAB 2-5
structure-like referencing 2-24
synchronous scheduling 3-10

T

target configuration options
    system target file 3-31
target preferences blocks in referenced models 3-47
timeout
    `timeout` 2-27
timer, configure 8-38
timer-based models, execution 3-10
timer-based scheduler 3-10
timing 3-9
tutorials
    objects for Green Hills MULTI® 2-2

W

write register 6-56