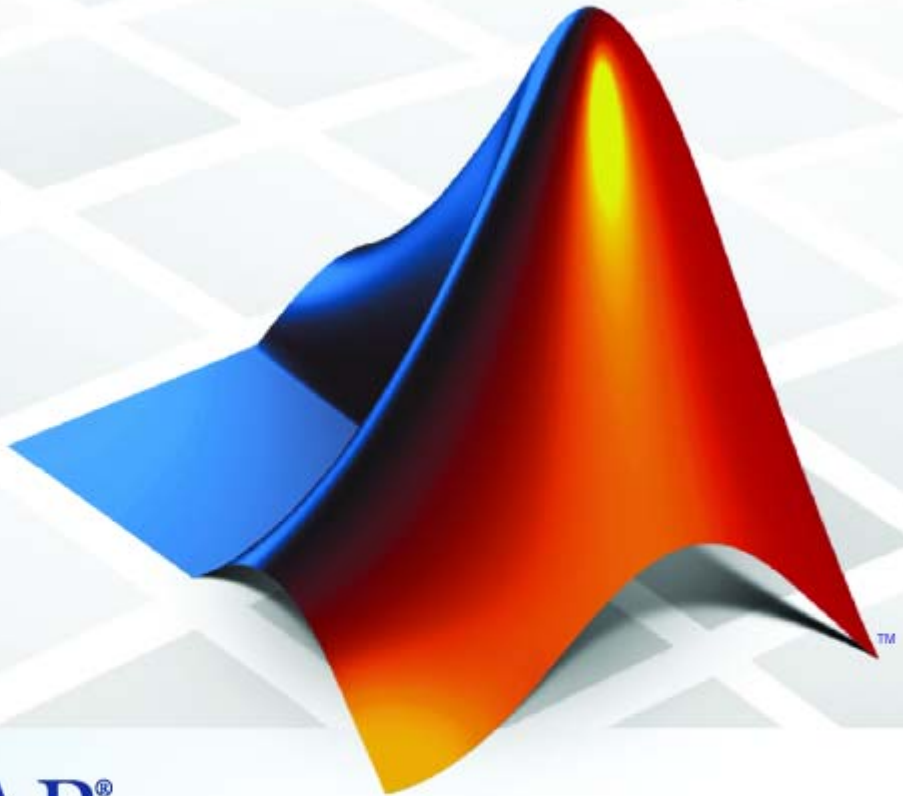


# Embedded IDE Link™ 4 User's Guide

*For Use with Analog Devices™ VisualDSP++®*



**MATLAB®**

## How to Contact The MathWorks



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



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*Embedded IDE Link™ User's Guide*

© COPYRIGHT 2007-2009 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

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

**Revision History**

May 2007	Online only	New for Version 1.0 (Release 2007a+)
September 2007	Online only	Revised for Version 1.1 (Release 2007b)
March 2008	Online only	Revised for Version 2.0 (Release 2008a)
October 2008	Online only	Revised for Version 2.1 (Release 2008b)
March 2009	Online only	Revised for Version 2.2 (Release 2009a)
September 2009	Online only	Revised for Version 4.0 (Release 2009b)



## Getting Started

### 1

<b>Product Overview</b> .....	1-2
<b>Software Structure and Components</b> .....	1-4
Automation Interface .....	1-4
Project Generator .....	1-5
Verification .....	1-5
<b>Software Requirements</b> .....	1-6

## Automation Interface

### 2

<b>Getting Started with Automation Interface</b> .....	2-2
Introducing the Automation Interface Tutorial .....	2-2
Running the Interactive Tutorial .....	2-5
Selecting Your Session and Processor .....	2-6
Querying Objects for VisualDSP++ IDE .....	2-7
Loading Files into VisualDSP++ IDE .....	2-9
Running the Project .....	2-10
Working with Global Variables and Memory .....	2-11
Working with Local Variables and Memory .....	2-13
Closing Files and Projects .....	2-15
Closing the Connections or Cleaning Up VisualDSP++ Software .....	2-16
Tutorial Summary .....	2-16
<b>Constructing Objects</b> .....	2-18
Example — Constructor for adivdsp Objects .....	2-18
<b>Properties and Property Values</b> .....	2-20

Setting and Retrieving Property Values .....	2-20
Setting Property Values Directly at Construction .....	2-21
Setting Property Values with set .....	2-21
Retrieving Properties with get .....	2-22
Direct Property Referencing to Set and Get Values .....	2-22
Overloaded Functions for adivdsp Objects .....	2-23
<b>adivdsp Object Properties .....</b>	<b>2-24</b>
Quick Reference to adivdsp Properties .....	2-24
Details About adivdsp Object Properties .....	2-25

## Project Generator

### 3

<b>Introducing Project Generator .....</b>	<b>3-2</b>
<b>Schedulers and Timing .....</b>	<b>3-3</b>
Configuring Models for Asynchronous Scheduling .....	3-3
Cases for Using Asynchronous Scheduling .....	3-4
Comparing Synchronous and Asynchronous Interrupt Processing .....	3-6
Using Synchronous Scheduling .....	3-8
Using Asynchronous Scheduling .....	3-8
Multitasking Scheduler Examples .....	3-9
<b>Project Generator Tutorial .....</b>	<b>3-22</b>
Building the Model .....	3-22
Adding the Target Preferences Block to Your Model .....	3-23
Specifying Simulink Configuration Parameters for Your Model .....	3-27
<b>Setting Code Generation Options for Analog Devices     Processors .....</b>	<b>3-30</b>
<b>Setting Real-Time Workshop Category Options .....</b>	<b>3-32</b>
Target File Selection .....	3-33
Build Process .....	3-33
Custom storage class .....	3-34

Report Options .....	3-34
Debug Pane Options .....	3-35
Optimization Pane Options .....	3-36
Embedded IDE Link Pane Options .....	3-38
Overrun Indicator and Software-Based Timer .....	3-43
Default Project Options — Custom .....	3-44

### **Optimizing Embedded Code with Target Function**

<b>Libraries</b> .....	3-46
About Target Function Libraries and Optimization .....	3-46
Using a Processor-Specific Target Function Library to Optimize Code .....	3-48
Process of Determining Optimization Effects Using Real-Time Profiling Capability .....	3-49
Reviewing Processor-Specific Target Function Library Changes in Generated Code .....	3-50
Reviewing Target Function Library Operators and Functions .....	3-52
Creating Your Own Target Function Library .....	3-52

<b>Model Reference</b> .....	3-53
How Model Reference Works .....	3-53
Using Model Reference .....	3-54
Configuring Targets to Use Model Reference .....	3-56

## **Verification**

# **4**

<b>What is Verification?</b> .....	4-2
<b>Verifying Generated Code via Processor-in-the-Loop</b> ..	4-3
What is Processor-in-the-Loop Cosimulation? .....	4-3
About the PIL Block .....	4-4
Preparing Your Model to Generate a PIL Application ....	4-5
Setting Model Configuration Parameters to Generate the PIL Application .....	4-6
Creating the PIL Block Application from a Model Subsystem .....	4-6

Running Your PIL Application to Perform Cosimulation and Verification .....	4-7
PIL Issues and Limitations .....	4-7
<b>Profiling Code Execution in Real-Time</b> .....	4-9
Overview .....	4-9
Profiling Execution by Tasks .....	4-10
Profiling Execution by Subsystems .....	4-13
<b>System Stack Profiling</b> .....	4-18
Overview .....	4-18
Profiling System Stack Use .....	4-19

## Function Reference

# 5

<b>Constructor</b> .....	5-2
<b>IDE Operations</b> .....	5-3
<b>Processor Operations</b> .....	5-4
<b>Debug Operations</b> .....	5-5
<b>Read/Write Operations</b> .....	5-6
<b>Get Information Operations</b> .....	5-7
<b>Object Information</b> .....	5-8
<b>Status Operations</b> .....	5-9
<b>Session Operations</b> .....	5-10
<b>Verification</b> .....	5-11



## Functions — Alphabetical List

6

## Block Reference

7

Block Library: idelinklib_aidvdsp .....	7-2
Block Library: idelinklib_common .....	7-3

## Blocks — Alphabetical List

8

## Configuration Parameters

9

<b>Embedded IDE Link Pane</b> .....	<b>9-2</b>
Overview .....	9-4
IDE link handle name .....	9-6
Profile real-time execution .....	9-7
Profile by .....	9-9
Number of profiling samples to collect .....	9-10
Project options .....	9-12
Compiler options string .....	9-14
Linker options string .....	9-16
System stack size (MAUs) .....	9-18
Build action .....	9-19
Interrupt overrun notification method .....	9-22
Interrupt overrun notification function .....	9-24
PIL block action .....	9-25
Maximum time allowed to build project (s) .....	9-27
Maximum time to complete IDE operations (s) .....	9-29
Source file replacement .....	9-31

## Reported Limitations and Tips

---

### A

<b>Reported Issues</b> .....	A-2
Using 64-bit Symbols in a 64-bit Memory Section on SHARC Processors .....	A-2

## Supported Processors

---

### B

<b>Supported Platforms</b> .....	B-2
Product Features Supported by Each Processor or Family .....	B-2
Supported Processors and Simulators .....	B-2
Custom Board Support .....	B-3

## Index

---

# Getting Started

---

- “Product Overview” on page 1-2
- “Software Structure and Components” on page 1-4
- “Software Requirements” on page 1-6

## Product Overview

Embedded IDE Link™ software provides a connection between MATLAB® and the VisualDSP++® IDE to enable you to access the processor from MATLAB. You can, manipulate data on the processor, and manage projects within the IDE, while simultaneously utilizing the MATLAB tools of numerical analysis and simulation. Using Embedded IDE Link software, you can perform the following tasks, and others related to Model-Based Design:

- Function calls — Write scripts in MATLAB software to execute any function in the VisualDSP++ IDE
- Automation — Write automated tests in MATLAB software to be executed on your processor, including control and verification operations
- Host-Processor Communication — Communicate with the processor directly from MATLAB software, without going to the IDE
- Verification and Validation
  - Load and execute projects into the VisualDSP++ IDE 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® software and run them on the processor
- Generate code— Generate executable code for your processor directly from the models designed in Simulink software, and execute it

Embedded IDE Link software connects MATLAB software and Simulink software with Analog Devices™ VisualDSP++® integrated development and debugging environment from Analog Devices™. Embedded IDE Link software enables you to use MATLAB and Simulink software to debug and verify embedded code running on all Analog Devices DSPs that VisualDSP++ software supports, such as the Analog Devices™ Blackfin®, Analog Devices™ SHARC® and Analog Devices™ TigerSHARC® processor families.

Embedded IDE Link software includes a project generator component. With the project generator component, you can generate a complete project for the VisualDSP++ IDE from your Simulink software models, including ANSI<sup>®</sup> C code generated with Real-Time Workshop<sup>®</sup> software. Thus, you use the Real-Time Workshop and Real-Time Workshop<sup>®</sup> Embedded Coder<sup>™</sup> software to generate generic ANSI C code projects for VisualDSP++ software from models. You can then build and run these projects on Blackfin<sup>®</sup>, SHARC<sup>®</sup>, and TigerSHARC<sup>®</sup> processors.

The following list suggests some of the uses for the capabilities of the software:

- Create test benches in MATLAB and Simulink software for testing your hand written or automatically generated code running on ADI DSPs
- Generate code and project files for VisualDSP++ software from Simulink models for rapid prototyping or deployment of a system or application
- Build, debug, and verify embedded code on ADI DSPs
- Perform processor-in-the-loop (PIL) testing of embedded code

## Software Structure and Components

In this section...
“Automation Interface” on page 1-4
“Project Generator” on page 1-5
“Verification” on page 1-5

Embedded IDE Link software comprises components—the Automation Interface component, the Project Generation component, and the Verification component. The Automation Interface component enables communication between MATLAB software and Embedded IDE Link software. The Project Generation component leverages Simulink software and lets you build models, simulate them, and generate code from the models directly to the processor.

The Verification component offers capabilities that help you use Model-Based Design to validate and verify your projects. With the Verification component, you can simulate algorithms and processes in Simulink models and concurrently on your processor. Comparing the results helps verify the fidelity of your model or algorithm code.

### Automation Interface

The Automation Interface component allows you to use Embedded IDE Link functions and methods to communicate with the VisualDSP++ IDE to perform the following tasks:

- Automate project management
- Debug programs
- Manipulate the data in the processor internal and external memory, and in the registers
- Communicate between the host and processor applications

The Debug Component of automation interface includes methods and functions for project automation, debugging, and data manipulation.

## Project Generator

The Project Generator component comprises methods that utilize the VisualDSP++ API to create projects in VisualDSP++ software and generate code with Real-Time Workshop and Real-Time Workshop Embedded Coder software. 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 software.
- Custom code generation — Use Embedded IDE Link software with any Real-Time Workshop System Target File (STF) to generate processor-specific and optimized code.
- Automatic downloading and debugging — Debug generated code in the VisualDSP++ debugger, using either the instruction set simulator or real hardware.
- Create and build projects for VisualDSP++ software from Simulink models — Project Generator uses Real-Time Workshop or Real-Time Workshop Embedded Coder software to build projects that work with Analog Devices processors.
- Generate custom code using the Configuration Parameters in your model with the system target files `vdspink_ert.tlc` and `vdspink_grt.tlc`.

## Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded IDE Link software combine to provide the following verification tools for you to apply as you develop your code:

### Processor-in-the-Loop Cosimulation

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

### Task Execution and Stack Usage Profiling

Gather execution profiling measurements with VisualDSP++ instruction set simulator to establish the timing requirements of your algorithm. Also, verify the stack usage is appropriate and as expected.

## Software Requirements

For detailed information about the software and hardware required to use Embedded IDE Link software, refer to the Embedded IDE Link system requirements areas on the MathWorks Web site:

- Requirements for Embedded IDE Link:  
[www.mathworks.com/products/ide-link/requirements.html](http://www.mathworks.com/products/ide-link/requirements.html)
- Requirements for use with VisualDSP++:  
[www.mathworks.com/products/ide-link/adi-adaptor.html](http://www.mathworks.com/products/ide-link/adi-adaptor.html)



# Automation Interface

---

- “Getting Started with Automation Interface” on page 2-2
- “Constructing Objects” on page 2-18
- “Properties and Property Values” on page 2-20
- “adivdsp Object Properties” on page 2-24

## Getting Started with Automation Interface

In this section...
“Introducing the Automation Interface Tutorial” on page 2-2
“Running the Interactive Tutorial” on page 2-5
“Selecting Your Session and Processor” on page 2-6
“Querying Objects for VisualDSP++ IDE” on page 2-7
“Loading Files into VisualDSP++ IDE” on page 2-9
“Running the Project” on page 2-10
“Working with Global Variables and Memory ” on page 2-11
“Working with Local Variables and Memory” on page 2-13
“Closing Files and Projects” on page 2-15
“Closing the Connections or Cleaning Up VisualDSP++ Software” on page 2-16
“Tutorial Summary” on page 2-16

### Introducing the Automation Interface Tutorial

Embedded IDE Link software provides a connection between MATLAB software and a processor in VisualDSP++ software. You can use 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 select a session in the VisualDSP++ IDE. The object you create is specific to a designated session in VisualDSP++ IDE.

---

To get you started using objects for VisualDSP++ software, Embedded IDE Link software includes an example script `vdsptutorial.m`. As you work

through this tutorial, you perform the following tasks that step you through creating and using objects for VisualDSP++ IDE.

- 1 Select your session.
- 2 Create and query objects to VisualDSP++ IDE.
- 3 Use MATLAB software to load files into VisualDSP++ software IDE.
- 4 Work with your VisualDSP++ IDE project from MATLAB software.
- 5 Close the connections you opened to VisualDSP++ IDE.

You use these tasks in any development work you do with signal processing applications. Thus, the tutorial provided here gives you a working process and best practice for using Embedded IDE Link software and your signal processing programs to develop programs for a range of Analog Devices processors.

The tutorial covers some methods and functions for Embedded IDE Link software. The functions listed first do not require an `adivdsp` object. The functions listed after that require an existing `adivdsp` object before you can use the function syntax.

### Functions for Working with VisualDSP++ Software

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

Function	Description
<code>listsessions</code>	Return information about the boards that VisualDSP++ IDE recognizes as installed on your PC.
<code>adivdsp</code>	Construct an object that refers to a VisualDSP++ IDE session. When you construct the object you specify the session by processor.

## Methods for Working with adivdsp Objects in VisualDSP++ Software

The following table presents some of the methods that require an adivdsp object.

<b>Methods</b>	<b>Description</b>
add	Add a file to a project
address	Return the address and page for an entry in the symbol table in VisualDSP++ IDE
build	Build the project in VisualDSP++ software
cd	Change the working directory
display	Display the properties of an object that references a VisualDSP++ software session
halt	Terminate execution of a process running on the processor
info	Return information about the object or session
isrunning	Test whether the processor is executing a process
load	Load a built project to the processor
open	Open a file in the project
read	Retrieve data from memory on the processor
reset	Restore the program counter (PC) to the entry point for the current program
run	Execute the program loaded on the processor
save	Save files or projects
visible	Set whether VisualDSP++ IDE window is visible on the desktop while VisualDSP++ IDE is running
write	Write data to memory on the processor

## Running VisualDSP++ Software on Your Desktop – Visibility

When you create an `adivdsp` object in the tutorial in the next section, Embedded IDE Link starts VisualDSP++ software in the background.

If VisualDSP++ software is running in the background, it does not appear on your desktop, in your task bar, or on the **Applications** page in the Task Manager. It does appear as a process, `idde.exe`, on the **Processes** tab in Task Manager.

You can make the VisualDSP++ IDE visible with the function `visible`. The function `isvisible` returns the status of the IDE—is it visible on your desktop. 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 `idde.exe`.

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

## 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 vdspautointtutorial`. 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 `vdspautointtutorial.m`.

---

**Note** To run the interactive tutorial, you must have at least one session configured in VisualDSP++ software. If you do not yet have a session, use the Analog Devices VisualDSP++ Configurator to create a session to use for this tutorial.

---

### Selecting Your Session and Processor

Embedded IDE Link IDE requires that you have at least one session available for VisualDSP++ software. To help you select the session to use for this tutorial, and for any development work, Embedded IDE Link software provides a command line tool, called `listsessions`, which prints a list of the available sessions. So that you can use this function in a script, `listsessions` can return a MATLAB structure that you use when you want your script to select a session in the IDE without your help.

---

**Note** The session you select is used throughout the tutorial.

---

- 1 To see a list of the sessions that you can use, enter the following command at the MATLAB prompt:

```
session_list = listsessions
```

MATLAB returns a list that shows all the sessions that Embedded IDE Link IDE recognizes as available in your installation.

```
session_list =  
  
    'ADSP-21060 ADSP-2106x Simulator'  
    'ADSP-21362 ADSP-2136x Simulator'
```

- 2 `listsessions` has a verbose mode that provides further details about the sessions in a cell array. The array contains structures that describe each session—the target type, the platform, and the processor.

```
sessionsinfo = listsessions('verbose');  
  
echo off
```

```
sessionname: 'ADSP-21362 ADSP-2136x Simulator'  
targettype: 'ADSP-2136x Family Simulator'  
platformname: 'ADSP-2136x Simulator'  
processors: 'ADSP-21362'
```

- 3 Use `adivdsp` to create an object that accesses a session in VisualDSP++ IDE.

```
vd = adivdsp('sessionname','ADSP-21362 ADSP-2136x Simulator','procnum',0)
```

`Sessionname` and `procnum` are property names that specify the property to set. `ADSP-21362 ADSP-2136x Simulator` is the session to access, and `0` is the number of the processor to refer to in the session.

When you use `adivdsp`, you create an object, in this case `vd`, that refers to the session you specify in `sessionname`.

## Querying Objects for VisualDSP++ IDE

In this tutorial section you create the connection between MATLAB and VisualDSP++ IDE. This connection, or object, is a MATLAB object, which for this session you save as variable `vd`. You use function `adivdsp` to create objects. When you create objects, `adivdsp` input arguments let you define other object properties, such as the global time-out. Refer to the `adivdsp` reference information for more about the input arguments.

Use the generated object `vd` to direct actions to your session processor. In the following tasks, `vd` appears in all function syntax that interact with IDE session and the processor: The object `vd` identifies and refers to a specific session. You need to include the object in any method syntax you use to access and manipulate a project or files in a session in VisualDSP++ IDE.

- 1 Create an object that refers to your selected session and processor. Enter the following command at the prompt.

```
vd = adivdsp('sessionname','ADSP-21362 ADSP-2136x Simulator','procnum',0)
```

If you watch closely, and your machine is not too fast, you see VisualDSP++ software appear briefly when you call `adivdsp`. If VisualDSP++ was not running before you created the new object, VisualDSP++ software starts and runs in the background.

Usually, you need to interact with VisualDSP++ while you develop your application. The function `visible`, controls the state of VisualDSP++ software on your desktop. `visible` accepts Boolean inputs that make VisualDSP++ software either visible on your desktop (input to `visible`  $\geq 1$ ) 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 VisualDSP++ IDE show on your desktop, enter the following command at the prompt:

```
visible(vd,1)
```

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

```
ADIVDSP Object:
  Session name      : ADSP-21362 ADSP-2136x Simulator
  Processor name    : ADSP-21362
  Processor type    : ADSP-21362
  Processor number  : 0
  Default timeout   : 10.00 secs
```

Embedded IDE Link software 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 Type `procinfo = info(vd)`.

The `vd` link status information provides data about the hardware, as follows:

```
procinfo =

  procname: 'ADSP-21362'
  proctype: 'ADSP-21362'
  revision: ''
```

- 5 Verify that the processor is running by entering



```
runstatus = isrunning(vd)
```

MATLAB responds, indicating that the processor is stopped, as follows:

```
runstatus =  
  
0
```

## Loading Files into VisualDSP++ IDE

In this part of the tutorial, you load the executable code for the CPU in the IDE. Embedded IDE Link software includes a tutorial project file for VisualDSP++ IDE. Through the next commands in the tutorial, you locate the tutorial project file and load it into VisualDSP++ IDE. The `open` method directs VisualDSP++ software 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 software cannot create a directory for you. If you do not have a writable directory, create one in Windows software before you proceed with the rest of this tutorial.

---

VisualDSP++ software has its own workspace and workspace files that are quite different from MATLAB workspace files and the MATLAB workspace. Remember to monitor both workspaces. The next steps change the working directory to your new writable directory.

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

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

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

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

```
cd(vd,prj_dir)
```

- 3** Next, use the following command to create a new VisualDSP++ software project named `dot_product_c.dpj` in the new directory:

```
new(vd, 'debug_demo.dpj')
```

Look in the IDE to verify that your new project exists. Next you need to add source files to your project.

- 4** Add the provided source file—`scalarprod.c` to the project `debug_demo.dpj` using the following command:

```
add(vd, [matlabroot '\toolbox\vdsp\link\vdspdemos\src\scalarprod.c'])
```

The variable `matlabroot` indicates the root directory of your MATLAB installation. Replace `matlabroot` with the path to MATLAB on your machine. For more information about the MATLAB root directory, refer to `matlabroot` in the MATLAB documentation.

- 5** Open the file in the IDE from MATLAB by issuing the following command to open the file:

```
open(vd, [matlabroot '\toolbox\vdsp\link\vdspdemos\src\scalarprod.c'])
```

Switch to the IDE to verify that the files are in your project and open.

- 6** Save your project.

```
save(vd, 'debug_demo.dpj', 'project')
```

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

## Running the Project

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

The next steps in this tutorial 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 `dot_product_c.dpj`.

```
build(vd,30) % The optional input argument 30 sets the time out period to 30 seconds.
```

At the end of the build process, Embedded IDE Link software returns a value of 1 to indicate that the build succeeded. If the build process returns a 0, the build failed.

```
ans =
     1
```

- 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.dxe`, and it is stored with the project in your writable directory, in a subdirectory named `debug`.

```
load(vd, 'c:\vdsp_demo\debug\debug_demo.dxe',30);
```

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

Try the following methods at the command prompt.

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

## Working with Global Variables and Memory

After you load your program on the processor, you can access memory locations and variables. You can then read variables either from the program symbol table or directly from addresses in memory. Three methods—`address`, `read`, and `write`, let you get, read, and write to and from your project and processor.

Start by getting the address of the global variable `v1` from the `debug_demo` project symbol table.

- 1 Enter the following command to retrieve the address for `v1`.

```
address_v1 = address(vd, 'v1')
```

```
address_v1 =
```

```
753666      1
```

- 2** Convert the address from decimal format to hexadecimal.

```
dec2hex(address_v1(1))
```

```
ans =
```

```
B8002
```

The address of global data array v1 is 0xB8002, which is stored in type 1 memory on the processor

- 3** With the address of v1 saved as address\_v1, use read to return the data from that location. To specify the data type and the number of values to read, add the datatype ('int32') and count (32) input arguments.

```
value_v1 = read(vd, address_v1, 'int32', 32) % Interpret the data as 32-bit integers.
```

```
value_v1 =
```

```
Columns 1 through 10
```

```
-37   -133    31  -104    32    66  -123    19   140   -28
```

```
Columns 11 through 20
```

```
16     80    -2    83  -243   148    56   163    46   45
```

```
Columns 21 through 30
```

```
-217   -11  -164   49    -3   99    21   -61   -26  101
```

```
Columns 31 through 32
```

```
-101   -151
```

- 4** Repeat the read process for another global variable in the project—`v2`. Nest the `address` method inside the `read` method to reduce typing.

```
value_v2 = read(vd,address(vd,'v2'),'int32',32) % Read and address methods in one call.
```

```
value_v2 =
```

```
Columns 1 through 10
```

```
    -50     5    -17    28     5    31    -23   -156     68    -5
```

```
Columns 11 through 20
```

```
   -220     5    -14    57    214    183    213     40    175    144
```

```
Columns 21 through 30
```

```
    -12    -77    -18    77    130    -39    132    107     52    -59
```

```
Columns 31 through 32
```

```
    127    -117
```

## Working with Local Variables and Memory

If you examine the source files for `debug_demo` in the IDE, you can verify the values for `v1` and `v2` in the source file `scalarprod.c`. You can also use the `address` method to get the addresses of local variables on the stack, after the variable is in scope.

To get the variables in scope (on the stack), you run the program. Adding a breakpoint to the program allows you to read the stack contents when the program stops at the breakpoint. Without the breakpoint, the program runs to completion, and you cannot read the contents of the stack because it no longer exists.

Begin the process by adding a breakpoint to the project file `scalarprod.c`:

- 1** Insert a breakpoint on line 100 of program `scalarprod.c` with the following command:

```
insert(vd, 'scalarprod.c', 100)
```

- 2** Run the program to add the variable to the stack, and move the program counter to the breakpoint. Add the optional input argument `timeout` sets the time out value to 30s instead of the default 20s value:

```
run(vd, 'runtohalt', 30)
```

The program stops at the breakpoint on line 100.

- 3** Read the address of the local variable `result`, and convert it to its hexadecimal equivalent value.

```
address_result = address(vd, 'result', 'local') % address_result is a 'local' variable.
```

```
address_result =
```

```
933884      1
```

```
dec2hex(address_result(1))
```

```
ans =
```

```
E3FFC
```

`address` returns `933884` as the location of `result` in memory, in type `1` memory on the processor, stored in the MATLAB variable `address_result`.

- 4** Use the variable `address_result` to get the value stored at that address by issuing the following `read` command:

```
actual_value_result = read(vd, address_result, 'int32')
```

```
actual_value_result =
```

```
18875
```

Verify in the IDE Output Window that `18875` is the correct value for the dot product.

- 5** Use the following command to remove the breakpoint set on line 100.

```
remove(vd, 'scalarprod.c', 100)
```

MATLAB includes a dot product function to use to verify the value in `actual_value_result`. Called `dot`, the function calculates the dot product of two input vectors. In this case, the inputs are vectors `value_v1` and `value_v2`.

Comparing the two results—`expected_value_result` in MATLAB with `actual_value_result` from the processor implementation validates your simulation and implementation. With Automation Interface methods, you can create MATLAB M-file scripts to test and verify algorithms in their implementation on a processor.

- 1 Calculate the expected result by performing the dot function with two input vectors.

```
expected_value_result = dot(value_v1, value_v2)
```

```
expected_value_result =
```

```
18875
```

- 2 Test to see if the actual and expected results match.

```
isequal(expected_value_result, actual_value_result)
```

```
ans =
```

```
1
```

- 3 After verifying the result and removing the breakpoint, run the program to completion, and then halt and reset the processor.

```
run(vd)
halt(vd)
reset(vd)
```

## Closing Files and Projects

You can close files in your projects from the MATLAB command line. The method `close` works at the command line to close programs or projects in the IDE through the `adivdsp` object and input keywords that describe the kind of file to close.

To finish this tutorial, close the open documents or files in the IDE, and then close the project `debug_demo.dpj`.

- 1 Close all of the open files and documents in the IDE. All of the open files are text files, so use the `text` input argument.

```
close(vd, 'all', 'text')
```

- 2 Now, close the project.

```
close(vd, 'debug_demo.dpj', 'project')
```

## Closing the Connections or Cleaning Up VisualDSP++ Software

Objects that you create in Embedded IDE Link software have connections to VisualDSP++ software. Until you delete these handles, the VisualDSP++ 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 `adivdsp` object, Embedded IDE Link software closes VisualDSP++ software. When it closes the IDE, the link software 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 `adivdsp` objects from your MATLAB workspace.

---

- 1 Use the following command to make the IDE invisible if it is visible on your desktop.

```
visible(vd.0)
```

- 2 To delete your connection to VisualDSP++ IDE, use `clear vd`.

## Tutorial Summary

During the tutorial you performed the following tasks:

- 1 Selected your session.



- 2** Created and queried objects that refer to a session in Embedded IDE Link to get information about the session and processor.
- 3** Used MATLAB to load files into VisualDSP++ IDE, and used methods in MATLAB to run that file.
- 4** Accessed variables in the program symbol table and on the processor.
- 5** Used the Automation Interface methods to compare the results of a simulation in MATLAB with the same algorithm running on a processor.
- 6** Closed the files, projects, and connections you opened to VisualDSP++ IDE.

## Constructing Objects

When you create a connection to a session in VisualDSP++ software using the `adivdsp` function, you create an object. The object implementation relies on MATLAB object-oriented programming capabilities similar to the objects you find in MATLAB or Filter Design Toolbox.

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

Like other MATLAB structures, objects in Embedded IDE Link software have predefined fields referred to as *object properties*.

You specify object property values by one of 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 `adivdsp` Objects

The easiest way to create an object is to use the function `adivdsp` to create an object with the default properties. Create an object named `vd` referring to a session in VisualDSP++ software by entering the following syntax:

```
vd = adivdsp
```

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

```
ADIVDSP Object:  
  Session name      : ADSP-21362 ADSP-2136x Simulator  
  Processor name    : ADSP-21362  
  Processor type    : ADSP-21362
```

```
Processor number : 0  
Default timeout  : 10.00 secs
```

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

---

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

---

## Properties and Property Values

In this section...
“Setting and Retrieving Property Values” on page 2-20
“Setting Property Values Directly at Construction” on page 2-21
“Setting Property Values with set” on page 2-21
“Retrieving Properties with get” on page 2-22
“Direct Property Referencing to Set and Get Values” on page 2-22
“Overloaded Functions for adivdsp Objects” on page 2-23

Objects in this 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 `adivdsp` 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 Embedded IDE Link software object property values with the `get` function.

Direct property referencing lets you either set or retrieve property values for `adivdsp` 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 `adivdsp`:

- 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 associated property value. Sometimes this value is also a string.

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

### Example — Setting Object Property Values at Construction

Suppose that you want to create a link to a session in VisualDSP++ software and set the following object properties:

- Refer to the specified session.
- Connect to the first processor.
- Set the global time-out to 5 s. The default is 10 s.

Set these properties by entering

```
vd = adivdsp('sessionname','ADSP-21060 ADSP-2106x Simulator','procnum',0,'timeout',5);
```

The `sessionname`, `procnum`, 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 change the value of any writable property of an object.

**Example — Setting Object Property Values Using set**

To set the time-out specification for the link `vd` from the previous section, enter the following syntax:

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

get(vd,'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 Object Property Values Using get**

To retrieve the value of the `sessionname` property for `vd2`, and assign it to a variable, enter the following syntax:

```
session = get(vd2,'sessionname')

session =

ADSP-21060 ADSP-2106x Simulator
```

**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.

```
vd = adivdsp;  
vd.time = 6;
```

## Overloaded Functions for adivdsp Objects

Several methods and functions in Embedded IDE Link 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 complete list of the methods that act on `adivdsp` objects, refer to the Chapter 6, “Functions — Alphabetical List” in the function reference pages.

## adivdsp Object Properties

### In this section...

“Quick Reference to adivdsp Properties” on page 2-24

“Details About adivdsp Object Properties” on page 2-25

Embedded IDE Link software provides links to your processor hardware so you can communicate with processors for which you are developing systems and algorithms. Because Embedded IDE Link software uses objects to create the links, the parameters you set are called properties and you treat them as properties when you set them, retrieve them, or modify them.

This section details the properties for the objects for VisualDSP++ software. First the section provides tables of the properties, for quick reference. Following the tables, the section offers in-depth descriptions of each property, its name and use, and whether you can set and get the property value associated with the property. Descriptions include a few examples of the property in use.

MATLAB users may find much of this handling of objects familiar. Objects in Embedded IDE Link software behave like objects in MATLAB and the other object-oriented toolbox products. C++ programmers may already understand the concepts described in this section.

### Quick Reference to adivdsp Properties

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

Property Name	User Settable?	Description
sessionname	At construction only	Reports the name of the session in VisualDSP++ IDE that the object references.



Property Name	User Settable?	Description
procnum	At construction only	Stores the number of the processor in the session. If you have more than one processor, this number identifies the specific processor.
timeout	Yes/default	Contains the global time-out 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 adivdsp Object Properties

To use the objects for VisualDSP++ interface, set values for the following:

- `sessionname` — Specify the session with which the object interacts.
- `procnum` — Specify the processor in the session. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — Specify the global time-out value. (Optional. Default is 10 s.)

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

### **procnum**

Property `procnum` identifies the processor referenced by an object for Embedded IDE Link IDE. Use `procnum` to specify the processor you are working with in the session specified by `sessionname`. The VisualDSP++ Configurator assigns a number to each processor installed in each session. To determine the value of `procnum` for a processor, use `listsessions` or the Configurator.

To identify a processor, you need the `sessionname` and `procnum` values. For sessions with one processor, `procnum` equals 0. VisualDSP++ IDE numbers the processors on multiprocessor boards sequentially from 0 to the total

number of processors. For example, on a board with four processors, the processors are numbered 0, 1, 2, and 3.

### **sessionname**

Property `sessionname` identifies the session referenced by a Embedded IDE Link software. When you create an object, you use `sessionname` to specify the session you are intending to interact with. To get the value for `sessionname`, use `listsessions` or the Analog Devices VisualDSP++ Configurator. The Configurator utility assigns the name for each session available on your system.

### **timeout**

Property `timeout` specifies how long VisualDSP++ software waits for any process to finish. You set the global time-out when you create an object for a session in VisualDSP++ IDE. The default global time-out value 10 s. The following example shows the `timeout` value for object `vd2`.

```
display(vd2)
```

```
ADIVDSP Object:
```

```
Session name      : ADSP-21060 ADSP-2106x Simulator  
Processor name    : ADSP-21060  
Processor type    : ADSP-21060  
Processor number  : 0  
Default timeout   : 10.00 secs
```

# Project Generator

---

- “Introducing Project Generator” on page 3-2
- “Schedulers and Timing” on page 3-3
- “Project Generator Tutorial” on page 3-22
- “Setting Code Generation Options for Analog Devices Processors” on page 3-30
- “Setting Real-Time Workshop Category Options” on page 3-32
- “Optimizing Embedded Code with Target Function Libraries” on page 3-46
- “Model Reference” on page 3-53

## Introducing Project Generator

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

- Automated project building for VisualDSP++ software that lets you create VisualDSP++ software projects from code generated by Real-Time Workshop and Real-Time Workshop Embedded Coder software. Project generator populates projects in the VisualDSP++ software development environment.
- Blocks in the library `idelinklib_adivdsp` for controlling the scheduling and timing in generated code.
- Highly configurable code generation using model configuration parameters and target preferences block options.
- Capability to use Embedded IDE Link software 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 software models to use the Project Generator component, do one or both of the following tasks:

- Add a Target Preferences block from the `idelinklib_adivdsp` library to the model.
- To use the asynchronous scheduler capability in Embedded IDE Link software, add one or more hardware interrupt blocks or idle task block from the `idelinklib_adivdsp` library.

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

## Schedulers and Timing

### In this section...

- “Configuring Models for Asynchronous Scheduling” on page 3-3
- “Cases for Using Asynchronous Scheduling” on page 3-4
- “Comparing Synchronous and Asynchronous Interrupt Processing” on page 3-6
- “Using Synchronous Scheduling” on page 3-8
- “Using Asynchronous Scheduling” on page 3-8
- “Multitasking Scheduler Examples” on page 3-9

### Configuring Models for Asynchronous Scheduling

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

- `idelinklib_common`

---

#### Note

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

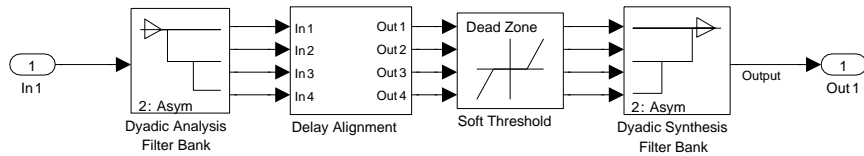
Also, you can schedule multiple tasks for asynchronous execution using the blocks.

The following figures show a model updated to use the asynchronous scheduler by converting the model to a function subsystem and then adding

a scheduling block (Hardware Interrupt) to drive the function subsystem in response to interrupts.

**Before**

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

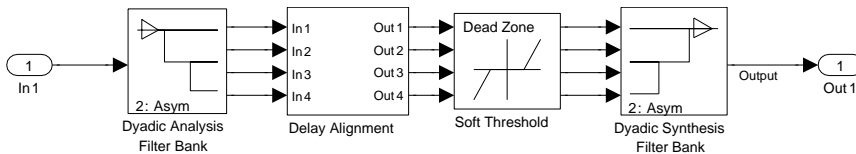


**After**

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

**Algorithm Inside the Function Call Subsystem Block**

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

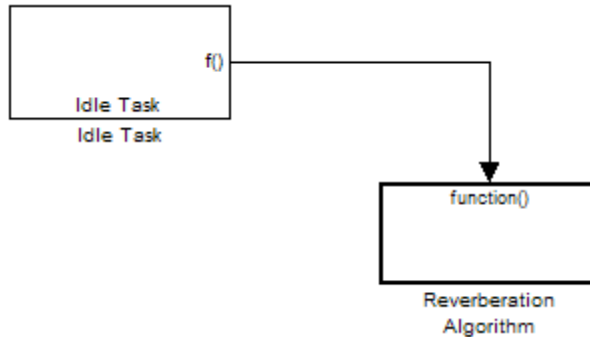


**Cases for Using Asynchronous Scheduling**

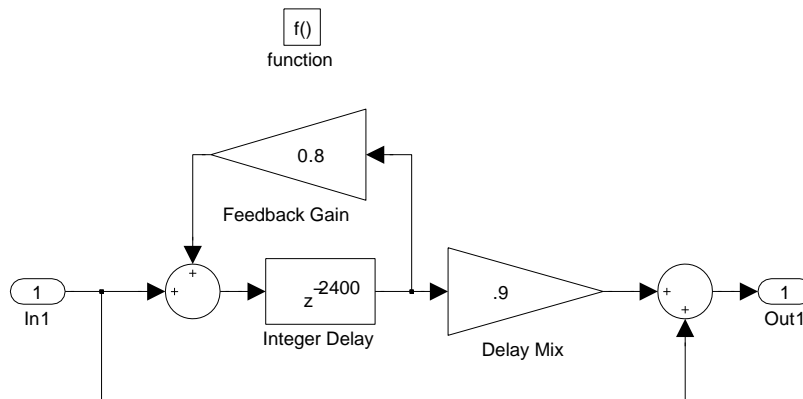
The following sections present common cases for using the scheduling blocks described in the previous sections.

## Idle Task

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



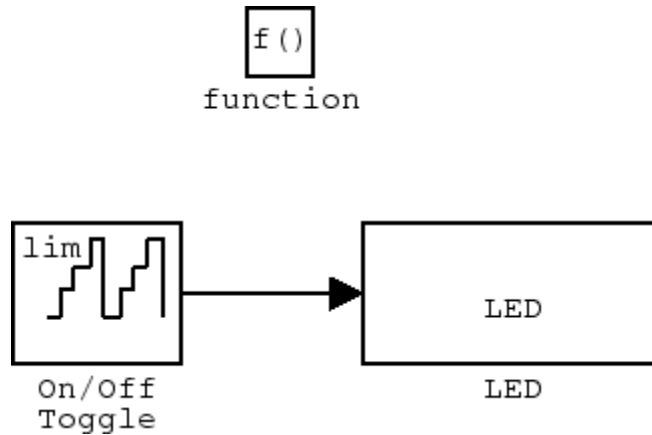
The function generated for this task normally runs in free-running mode—repetitively and indefinitely. Subsystem execution of the reverberation function is data driven via a background DMA interrupt-controlled ISR, shown in the following figure.



## Hardware Interrupt Triggered Task

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

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



## Comparing Synchronous and Asynchronous Interrupt Processing

Code generated for periodic tasks, both single- and multitasking, runs via a timer interrupt. A timer interrupt ensures that the generated code representing periodic-task model blocks runs at the specified period. The periodic interrupt clocks code execution at runtime. This periodic interrupt clock operates on a period equal to the base sample time of your model.

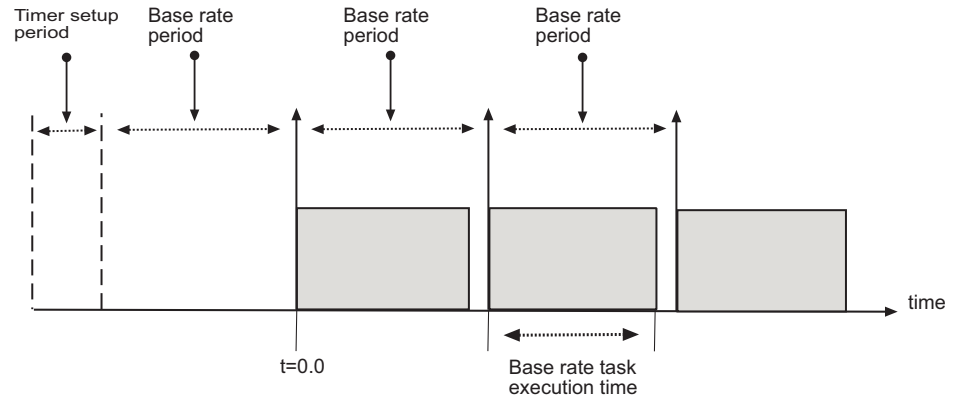
---

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

---

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





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

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

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

---

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

---

## Using Synchronous Scheduling

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

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

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

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

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

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

## Using Asynchronous Scheduling

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

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

The Hardware Interrupt block operates by

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

---

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

---

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

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

## Multitasking Scheduler Examples

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

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

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

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

Examples in this section demonstrate a variety of multitasking configurations:

- “Three Odd-Rate Tasks Without Preemption and Overruns” on page 3-12

- “Two Tasks with the Base-Rate Task Overrunning, No Preemption” on page 3-13
- “Two Tasks with Sub-Rate 1 Overrunning Without Preemption” on page 3-14
- “Three Even-Rate Tasks with Preemption and No Overruns” on page 3-15
- “Three Odd-Rate Tasks Without Preemption and the Base and Sub-Rate1 Tasks Overrun” on page 3-17
- “Three Odd-Rate Tasks with Preemption and Sub-Rate 1 Task Overruns” on page 3-18
- “Three Even-Rate Tasks with Preemption and the Base-Rate and Sub-Rate 1 Tasks Overrun” on page 3-20

Each example presents either two or three tasks:

- **Base Rate task.** Base rate is the highest rate in the model or application. The examples use a base rate of 1ms so that the task should execute every one millisecond.
- **Sub-Rate 1.** The first subrate task. Sub-Rate 1 task runs more slowly than the Base-Rate task. Sub-Rate 1 task rate is 2ms in the examples so that the task should execute every 2ms.
- **Sub-Rate 2.** In examples with three tasks, the second subrate task is called Sub-Rate 2. Sub-Rate 2 tasks run more slowly than Sub-Rate 1. In the examples, Sub-Rate 2 runs at either 4ms or 3ms.
  - When Sub-Rate 2 is 4ms, the example is called *even*.
  - When Sub-Rate 2 is 3ms, the example is called *odd*.



---

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

---

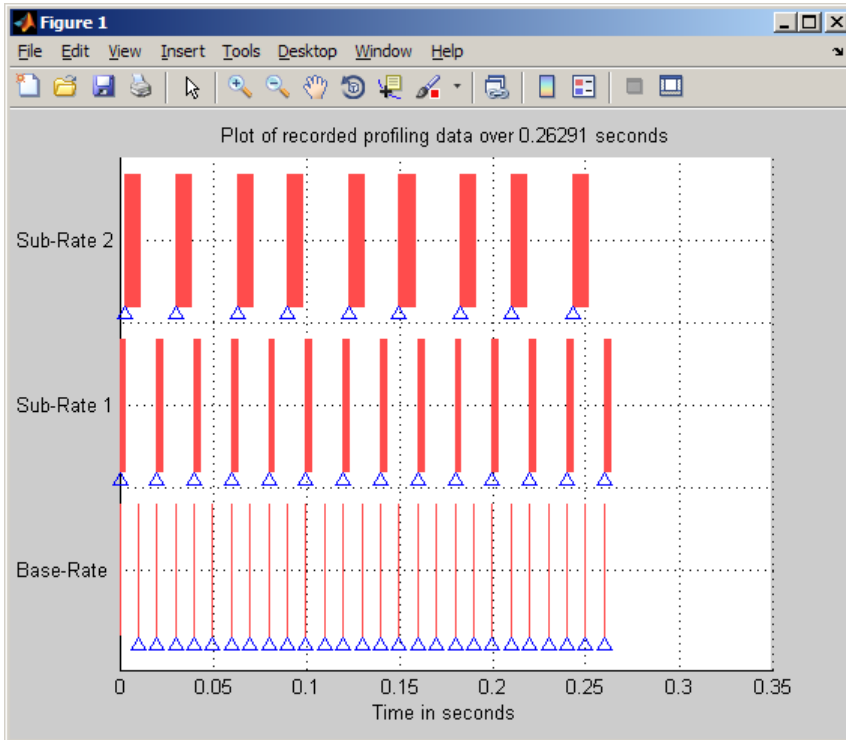
The following legend applies to the plots in the next sections:

- Blue triangles (  ) indicate when the task started.

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

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

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

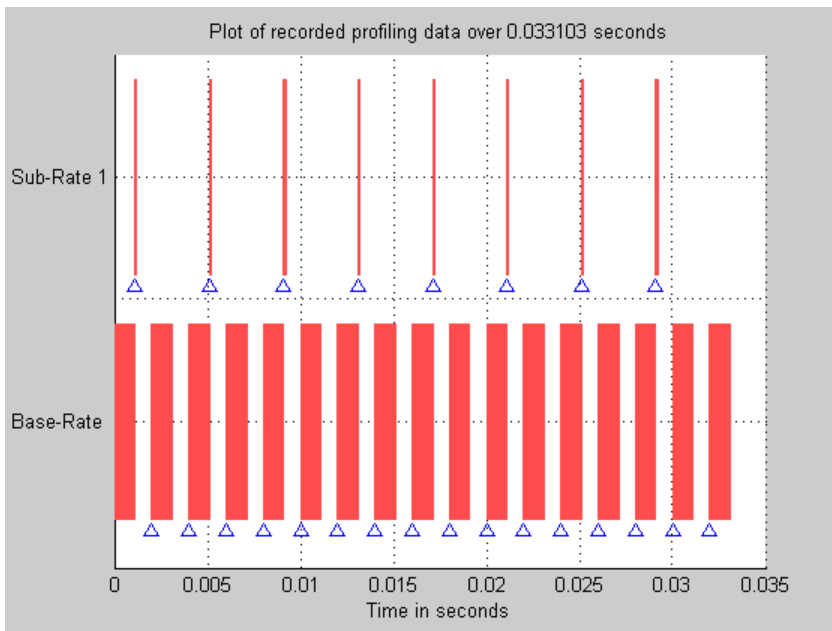


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	1ms
Sub-Rate 1	2ms	2ms
Sub-Rate 2	3ms	3ms

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

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

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



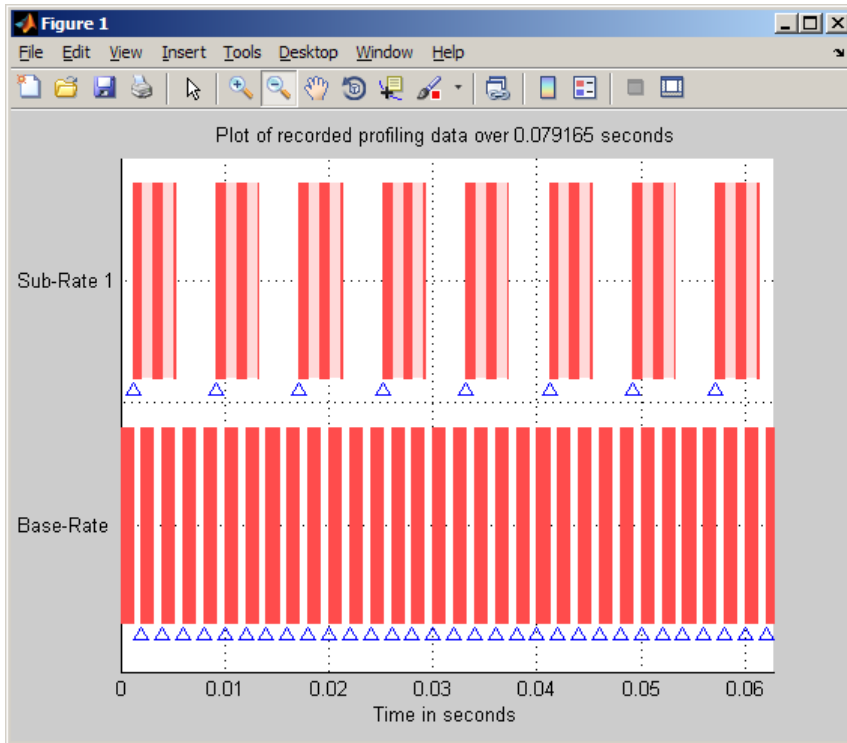
Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)

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

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

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



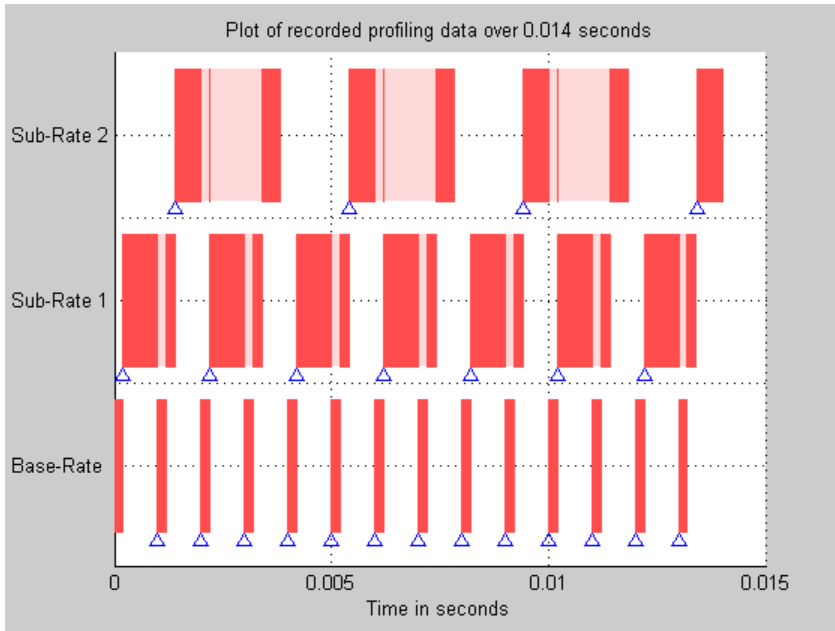


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	8ms (overrunning)

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

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

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

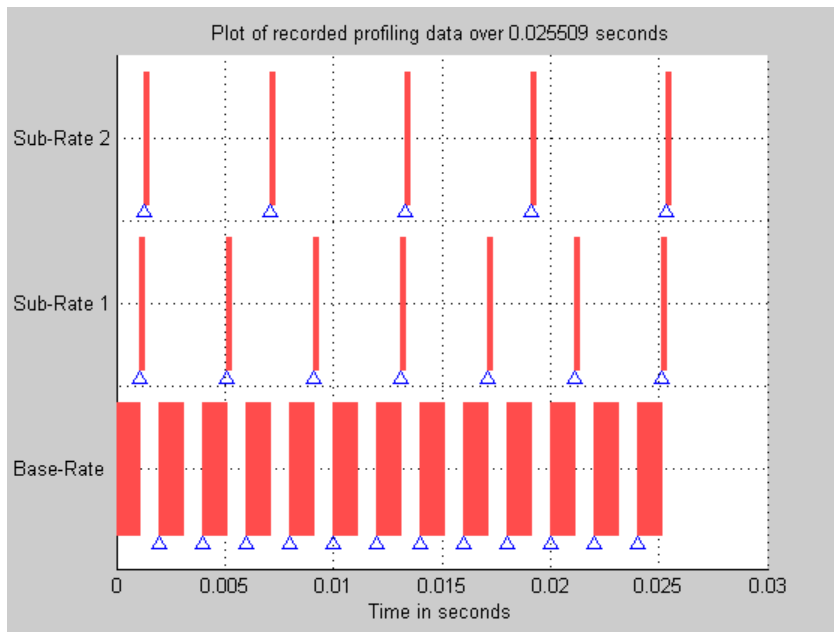


<b>Task Identification</b>	<b>Intended Execution Schedule</b>	<b>Actual Execution Schedule</b>
Base-Rate	1ms	1ms
Sub-Rate 1	2ms	2ms
Sub-Rate 2	4ms	4ms

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

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

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

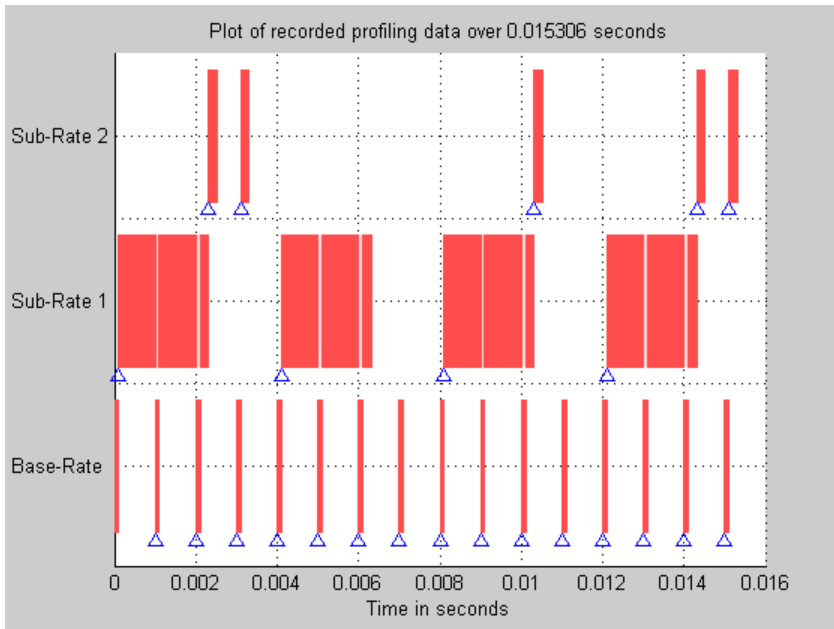


Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)
Sub-Rate 1	2ms	4ms (overrunning)
Sub-Rate 2	3ms	6ms (overrunning)

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

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

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



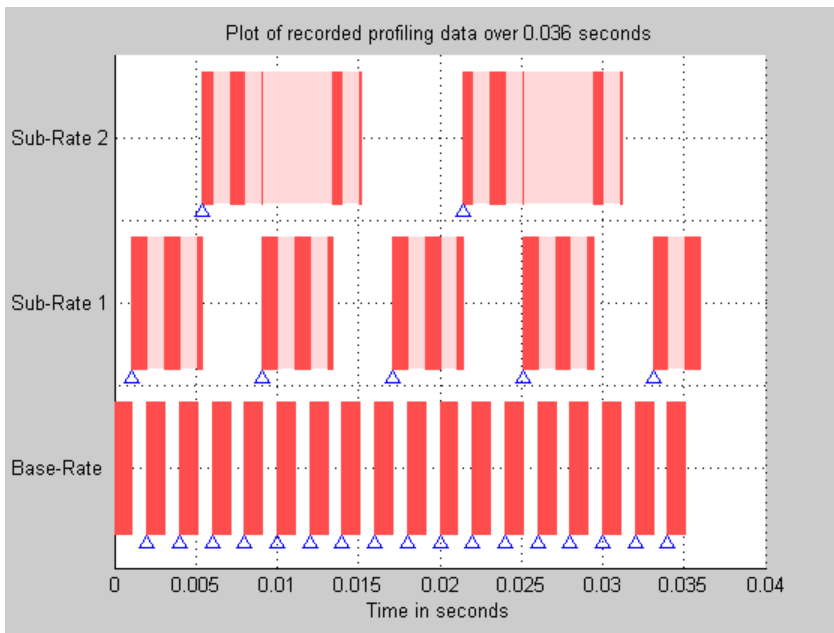
Task Identification	Intended Execution Schedule	Actual Execution Schedule
Base-Rate	1ms	2ms (overrunning)

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

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

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

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



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

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

## Project Generator Tutorial

### In this section...

“Building the Model” on page 3-22

“Adding the Target Preferences Block to Your Model” on page 3-23

“Specifying Simulink Configuration Parameters for Your Model” on page 3-27

In this tutorial you build a model and generate a project from the model into VisualDSP++ IDE.

---

**Note** The model demonstrates 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:

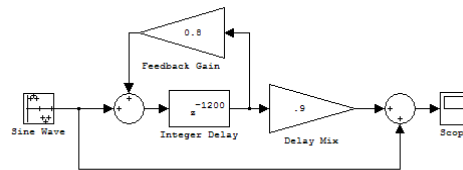
- 1** Use Simulink blocks, Signal Processing Blockset™ blocks, and blocks from other blocksets to create the model application.
- 2** Add the target preferences block from the Embedded IDE Link Target Preferences library to your model. Verify and set the block parameters for your hardware. In most cases, the default settings work fine.
- 3** Set the configuration parameters for your model, including the following parameters:
  - Solver parameters such as simulation start and solver options
  - Real-Time Workshop software options such as processor configuration and processor compiler selection
- 4** Generate your project.
- 5** Review your project in VisualDSP++ software.

### Building the Model

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



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



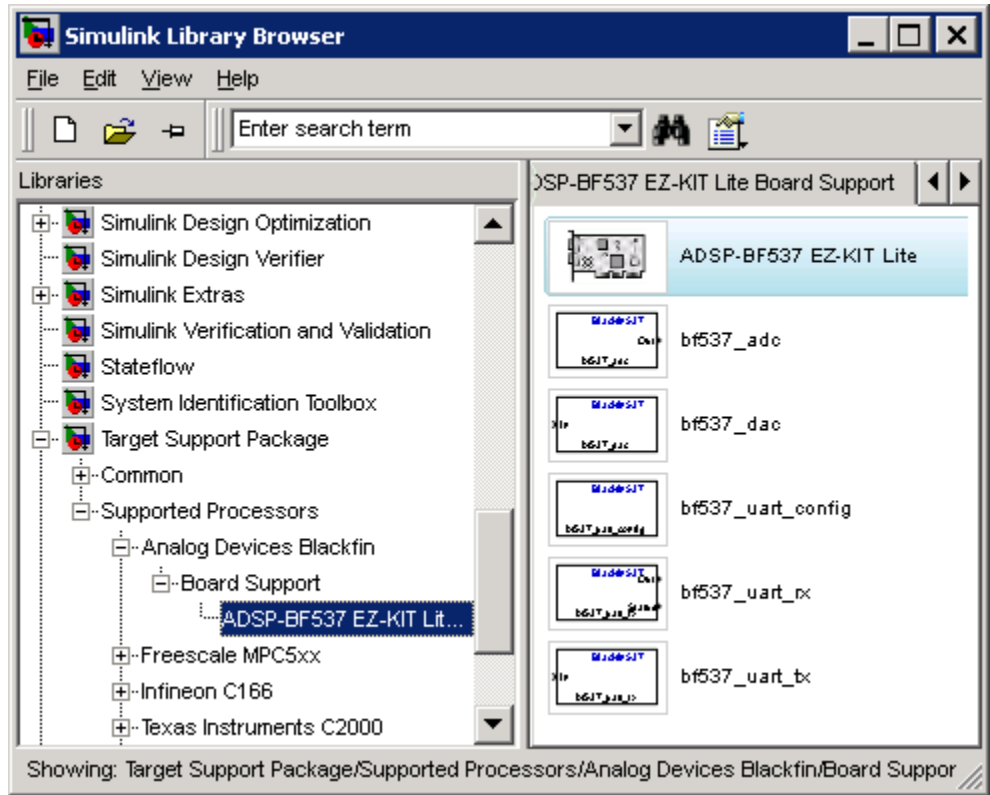
Look for the Integer Delay block in the Discrete library of Simulink and the Gain block in the Commonly Used Blocks library. Do not add the Custom Board block at this time.

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

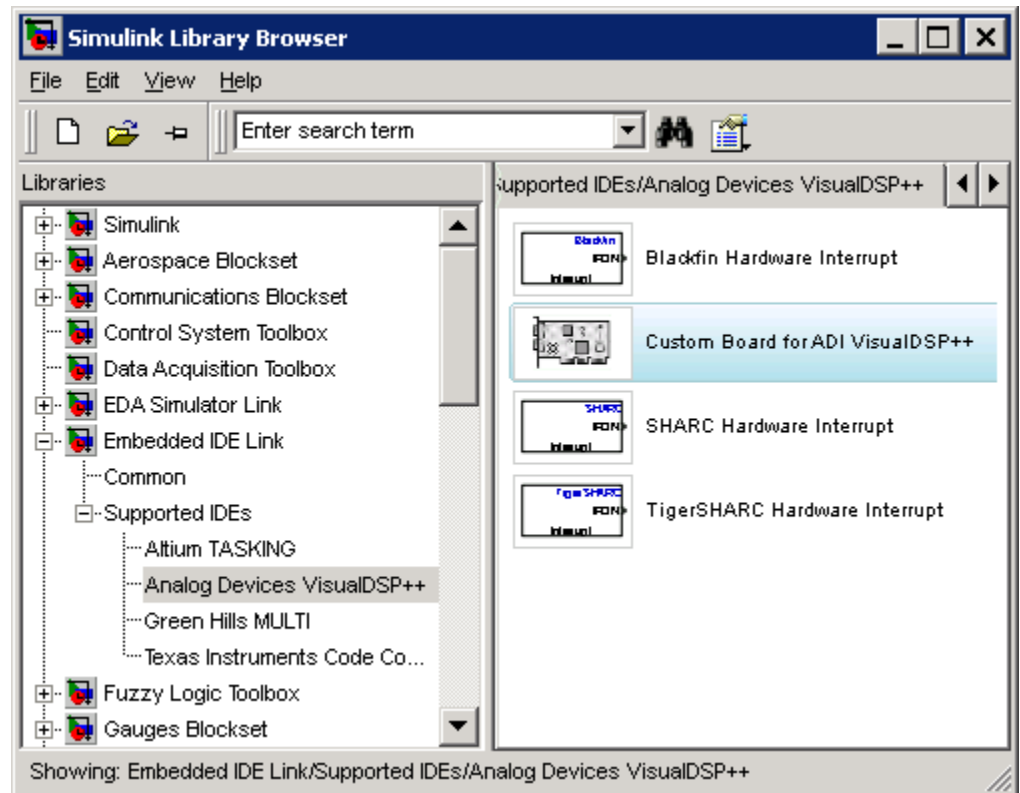
## Adding the Target Preferences Block to Your Model

To configure your model to work with Analog Devices processors, add a target preferences block to your model.

If you have Target Support Package™ software, check the list of supported processors in the Simulink library browser for a pre-configured Target Preferences block for your processor. For example:

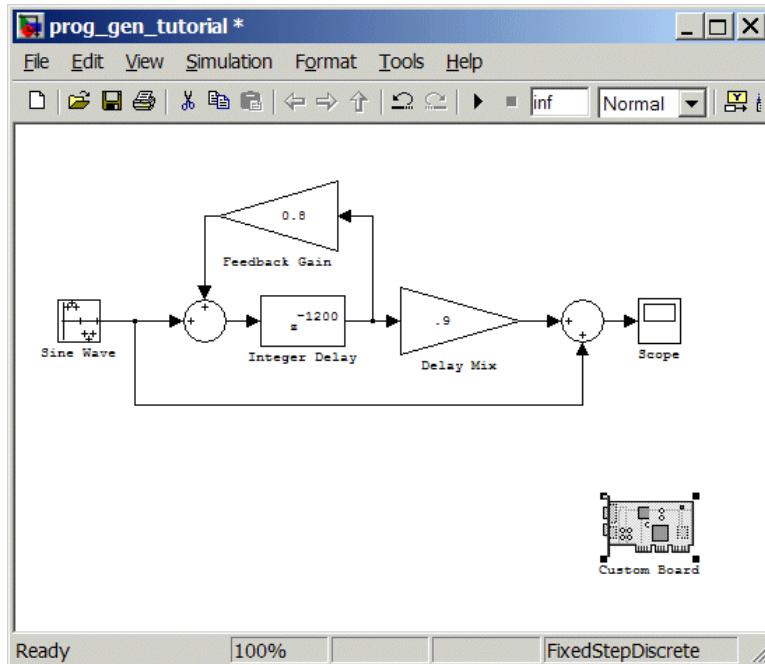


Otherwise, use the Target Preferences/Custom Board for ADI VisualDSP++ block, located in the `idelinklib_adi_vdsp` block library.



To configure the Target Preferences/Custom Board for ADI VisualDSP++ (the “Custom Board”) block in your model:

- 1 Drag and drop the Custom Board block to your model as shown in the following figure.



- 2 Double-click the Custom Board block to open the block dialog box.
- 3 In the block dialog box, select your processor from the **Processor** list.
- 4 Verify the **CPU clock** value.
- 5 Select the session name from the **Session name** list. Verify that the session processor matches the one you selected from the **Processor** list.
- 6 Review the settings on the **Memory** and **Sections** tabs to verify that they are correct for the processor you selected.
- 7 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 VisualDSP++ IDE 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 software.

### Setting Solver Options

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

- 1 Open the Configuration Parameters dialog box and set the appropriate options on the **Solver** category for your model and for Embedded IDE Link software.
  - Set **Start time** to 0.0 and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to `inf` for completeness.
  - Under **Solver options**, select the fixed-step and discrete settings from the lists when you generate executable projects. When you use PIL, use any setting on the **Type** and **Solver** 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 Software Options

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

- 1 Select Real-Time Workshop on the **Select** tree.
- 2 In Target selection, click **Browse** to select the system target file for Analog Devices processors—`vdsplink_grt.tlc`. It may already be the selected target file.

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 system target file `vdsplink_grt.tlc`, and click **OK** to close the browser.

## Setting Embedded IDE Link Options

After you set the Real-Time Workshop options for code generation, set the options that apply to your Analog Devices processor.

- 1 Change the category on the **Select** tree to Hardware Implementation.
- 2 Verify that the Device type is the correct value for your processor—ADI Blackfin, ADI SHARC, or ADI TigerSHARC.
- 3 From the **Select** tree, choose Embedded IDE Link to specify code generation options that apply to the processor.
- 4 Under **Code Generation**, clear all of the options.
- 5 (optional) Under **Link Automation**, provide a name for the handle in **IDE handle name**.
- 6 Set the following options in the dialog box under **Project options**:
  - Set **Project options** to Custom.
  - Set **Compiler options string** and **Linker options string** to blank.

**7** Set the following **Runtime** options:


- **Build action:** Create\_project.
- **Interrupt overrun notification method:** Print\_message.

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 software. In some cases, you may decide to set options in the other categories.

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

## Creating Your Project

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

- 1** Click **OK** to close the Configuration Parameters dialog box.
- 2** Click Incremental Build () on the model toolbar to generate your project into VisualDSP++ IDE.

When you click  with Create\_project selected for **Build action**, the automatic build process starts VisualDSP++ software and populates a new project in the development environment.

## Setting Code Generation Options for Analog Devices Processors

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

---

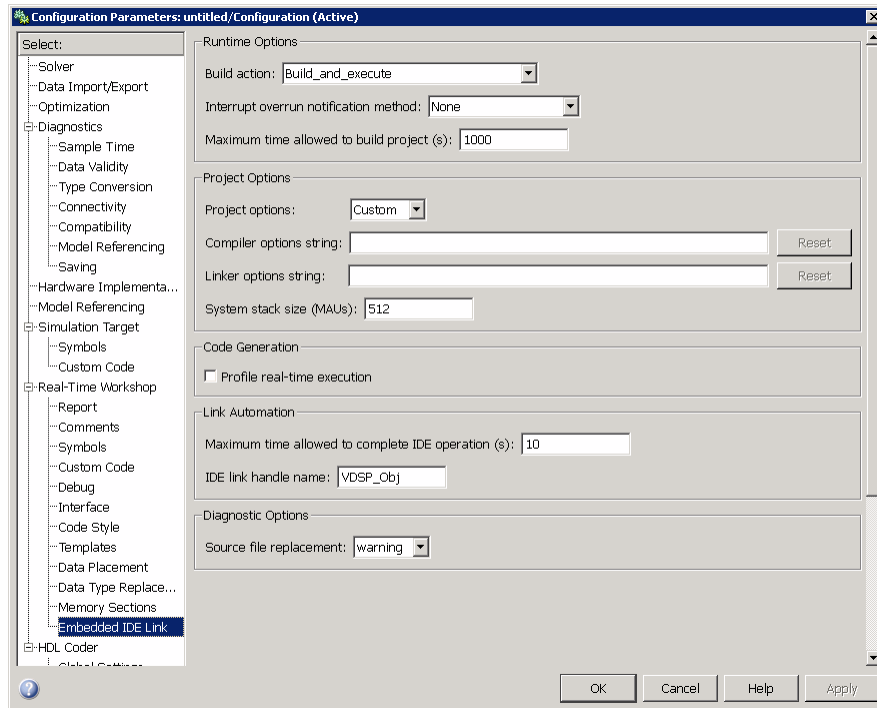
**Note** Embedded IDE Link software does not support continuous states in Simulink software models for code generation. In the **Solver options** in the Configuration Parameters dialog box, you must select **Discrete (no continuous states)** as the **Type**, along with **Fixed step**. When you use PIL,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

---

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 software.





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 including the processor and always appear on the list.

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

Set **System target file** to `vdspLink_grt.tlc` or `vdspLink_ert.tlc`. This adds “Embedded IDE Link” to the **Select** tree. The `vdspLink_grt.tlc` file is appropriate for all projects. Use the `vdspLink_ert.tlc` to develop projects or code for embedded processors (requires Real-Time Workshop Embedded Coder software) or you plan to use Processor-in-the-Loop features.

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

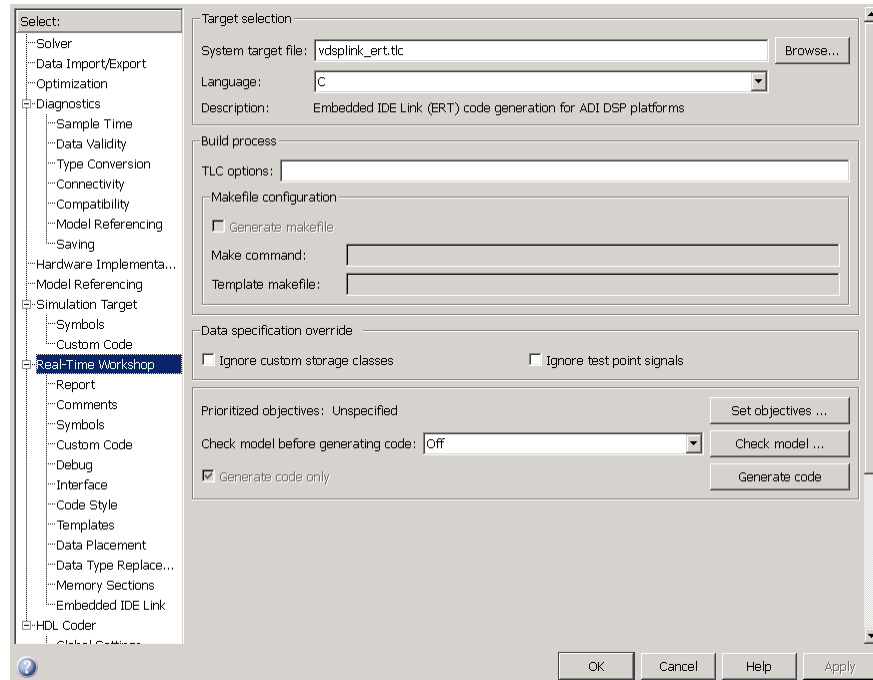
## Setting Real-Time Workshop Category Options

In this section...
“Target File Selection” on page 3-33
“Build Process” on page 3-33
“Custom storage class” on page 3-34
“Report Options” on page 3-34
“Debug Pane Options” on page 3-35
“Optimization Pane Options” on page 3-36
“Embedded IDE Link Pane Options” on page 3-38
“Overrun Indicator and Software-Based Timer” on page 3-43
“Default Project Options — Custom” on page 3-44

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

- Determine your processor, either Analog Devices or some other processor if you are not using Embedded IDE Link software.
- Configure your build process.
- Specify whether to use custom storage classes.

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



## Target File Selection

### System target file

Clicking **Browse** opens the Target File Browser where you select `vdsplink_grt.tlc` as your Real-Time Workshop **System target file**.

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

### Build Process

Embedded IDE Link software does not use makefiles or the build process to generate code. Code generation is project based so the options here do not apply.

## Custom storage class

When you generate code from a model employing custom storage classes (CSC), clear **Ignore custom storage classes**. This setting is the default value for Embedded IDE Link 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 software. To generate source code without building and executing the code on your processor, select Embedded IDE Link from the **Select** tree. Then, under **Runtime**, select `Create_project` for **Build action**.

## Report Options

Two options control HTML report generation during code generation.

- “Create Code Generation report” on page 3-34
- “Launch report automatically” on page 3-35

## Create Code Generation 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 'Create code generation report'
```

In the Navigation options, when you select **Model-to-code** and **Code-to-model**, your HTML report includes hyperlinks to various features in your Simulink model.

### Launch report automatically

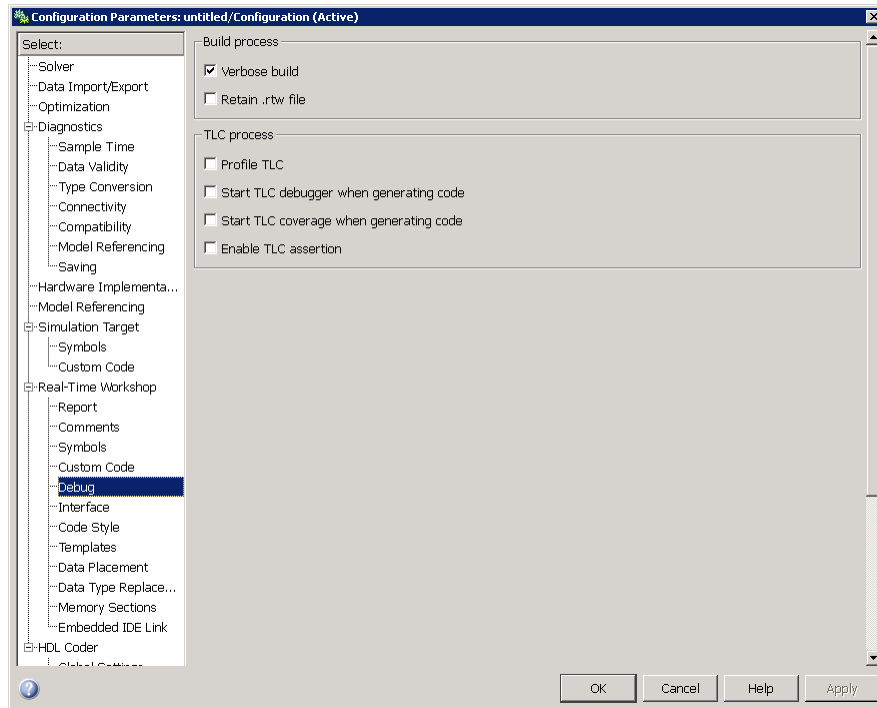
This option directs Real-Time Workshop to open a MATLAB Web browser window and display the code generation report. If you clear this option, you can open the code generation report (*modelName\_codegen\_rpt.html* or *subsystemname\_codegen\_rpt.html*) manually in a MATLAB Web browser window or in another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

## Debug Pane Options

Real-Time Workshop software 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.

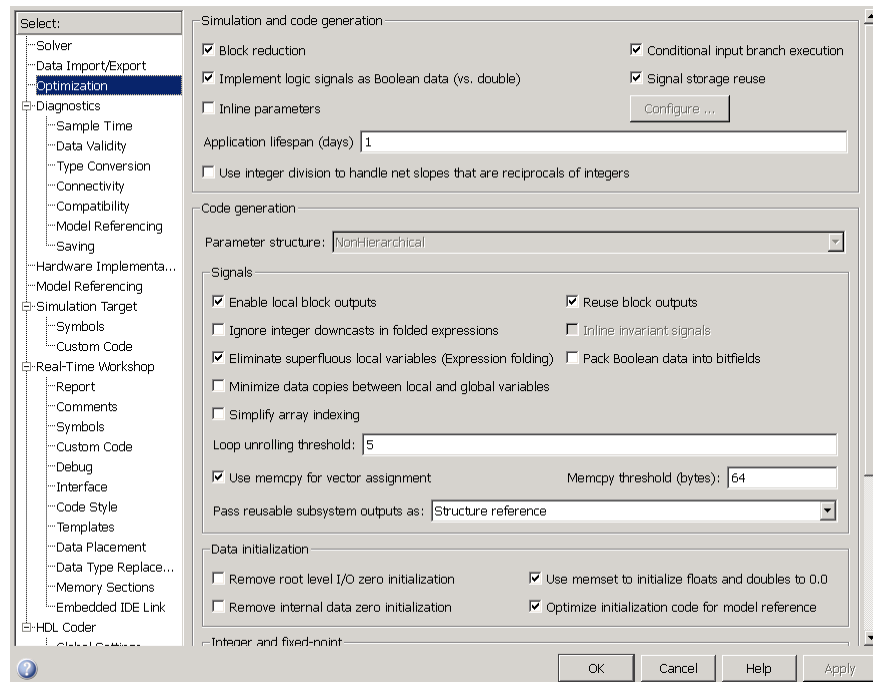
When you select **Debug** from the **Select** tree, you see the **Debug** options as shown in the next figure. In this pane, you set options that are specific to the Real-Time Workshop software process and TLC debugging.



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 software 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 `vdsp1ink_grt.tlc` under **Real-Time Workshop system target file**.



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

- **Conditional input branch execution**
- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse block outputs**
- **Eliminate superfluous temporary variables (Expression folding)**
- **Loop unrolling threshold**
- **Optimize initialization code for model reference**

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

## Embedded IDE Link Pane Options

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

- **Runtime** — Set options for run-time operations, like the build action.
- **Project Options** — Set build options for your project code generation, including compiler and linker settings.
- **Code Generation** — Configure your code generation requirements, such as enabling real-time task execution profiling.
- **Link Automation** — Export a handle to your MATLAB workspace.
- **Diagnostic Options** — Set options that control code generation diagnostic messages.

### Runtime

Before you run your model as an executable on any Analog Devices processor, you must configure the run-time options for the model.

By selecting values for the options available, you configure the operation of your model build process and overrun handling.

### Build action

To specify to Real-Time Workshop software what to do when you click **Build**, select one of the following options.:

- **Create\_project** — Directs Real-Time Workshop software to start VisualDSP++ IDE and populate a new project with the files from the build process. This option offers a convenient way to build projects in VisualDSP++ software.
- **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 VisualDSP++ software projects for models that you use in model referencing.
- **Build** — Builds the executable file, but does not download the file to your processor.



- `Build_and_execute` — Directs Real-Time Workshop software to build, download, and run your generated code as an executable on your processor.
- `Create_processor_in_the_loop_project` — Directs the Real-Time Workshop code generation process to create PIL algorithm object code as part of the project build.

Your selection for **Build action** determines what happens when you click **Build** or press **Ctrl+B**. Your selection tells Real-Time Workshop software at which stage 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 software 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. You do not need to reset the board before building models.

---

### **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 should use to notify you that an overrun condition occurred. The function must exist in your code on the processor.

### **PIL block action**

If you have Real-Time Workshop Embedded Coder software installed and you select the `vdspink_ert.tlc` system target file, you can choose to use the processor-in-the-loop (PIL) feature provided by Embedded IDE Link software. Selecting `Create_processor_in_the_loop_project` for the **Build action** enables the **PIL block action** option. **PIL block action** specifies whether Real-Time Workshop software builds the PIL block and downloads the block to the processor.

Choose one of the following three actions for creating a PIL block:

- **None** — Configures model to generate a VisualDSP++ project that contains the PIL algorithm code. Does not build the PIL object code or block. The new project will not compile in VisualDSP++ software.
- **Create PIL block** — Creates a PIL block, places the block in a new model, and then stops without building or downloading the block. The resulting project will not compile in VisualDSP++ IDE.
- **Create PIL block\_build\_and\_download** — Builds and downloads the PIL application to the processor after creating the PIL block. Adds PIL interface code that exchanges data with Simulink software. Use this selection to update the algorithmic code in an existing PIL block in a model.

Your selections affect how you use the resulting PIL block. The following list describes the build process and actions you take based on the **PIL block action** setting:

- When you click **Build** on the PIL dialog box, the build process adds the PIL interface code to the project and compiles the project in VisualDSP++ IDE.
- If you select **Create PIL block**, you can build manually from the right-click context menu on the PIL block.
- After you select **Create PIL Block**, copy the PIL block into your model to replace the original subsystem. Save the original subsystem in a different model so you can restore in the future. Click **Build** to build your model with the PIL block in place.
- Add the PIL block to your model to use cosimulation to compare PIL results with the original subsystem results. Refer to the demo Getting Started with Application Development in the product demos Embedded IDE Link

- To use the PIL block in a project after you selected None or Create PIL block for **Block action** when you built the project, click **Build** followed by **Download** in the PIL block dialog box.

The following table summarizes the effects of the PIL block action options.

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.

### Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message. 1000 s is the default to allow extra time to complete project builds and code generation.

### 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 software specified compiler and linker settings for your generated code.

### Compiler options string

To let you determine the degree of optimization provided by the Analog Devices optimizing compiler, you enter the optimization level to apply to files in your project. For details about the compiler options, refer to your VisualDSP++ documentation. When you create new projects, Embedded IDE Link does not set optimization options.

### Linker options string

To let you specify the options provided by the Analog Devices linker during link time, you enter the linker options as a string. For details about the linker options, refer to your VisualDSP++ documentation. When you create new projects, Embedded IDE Link software does not set linker options.

### System stack size (MAUs)

Enter the amount of memory to use for the stack. For more information on memory requirements, refer to **Enable local block outputs** on the **Optimization** pane of the Configuration Parameters dialog box. Block output buffers are placed on the stack until the stack memory is fully allocated. 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 software options for configuring and building models and generating code.

### Code Generation

From this category, you select options that define the code generation process.

To enable the real-time execution profile capability, select **Profile real-time 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 both a graphical presentation and an HTML report form. For more information, see “Profiling Execution by Tasks” on page 4-10 and “Profiling Execution by Subsystems” on page 4-13

### Link Automation

When you use Real-Time Workshop software to build a model to an Analog Devices processor, Embedded IDE Link software makes a connection between MATLAB software and the VisualDSP++ IDE. MATLAB software represents that connection as an `adivdsp` object. The properties of the `adivdsp` object contain information about the IDE instance it refers to, such as the session and processor it accesses. In this pane, the **IDE link handle name** option instructs Embedded IDE Link software to export the `adivdsp` object created during code generation to your MATLAB workspace with the name you enter. Replace the default name `VDSP_obj` with your own name for the object to export.

### **Maximum time to complete IDE operations (s)**

Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages.

### **Diagnostic Options**

When you generate code from a model, the options in this section determine what diagnostic messages you see and how the build process responds to the diagnostics.

### **Source file replacement**

Selects the diagnostic action to take if Embedded IDE Link software detects conflicts when you replace source code with custom code.

The following information can help you use the diagnostic messages in your work.

- The build operation continues if you select `warning` and the software detects custom code replacement problems. You see warning messages as the build progresses.
- Use the `error` setting the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Use `none` when you are sure the replacement process is correct and do not want to see multiple messages during your build.

### **Overrun Indicator and Software-Based Timer**

When your digital signal process application cannot complete the calculations and data manipulations required to yield a result before the available clock cycles expire, your model can generate unreliable data. Failing to complete an algorithm is called `overrunning`, and is one of the most important errors to identify and eliminate in digital signal processing design and implementation.

The Embedded IDE Link software provides an overrun indicator that notifies you when your process overruns.

Limitations — The overrun indicator does not work in multirate systems where the rate in the process is not the same as the base clock rate for your

model. When this is the case, the timer/scheduler in the DSP provides the interrupts for setting the model rate and cannot indicate that an overrun has occurred.

### **Default Project Options – Custom**

Although VisualDSP++ software offers standard project configurations, **Release** and **Debug**, models you build with Embedded IDE Link software use **Custom** for a custom configuration that provides a third combination of build and optimization.

Project configurations define sets of project build options. When you specify the build options at the project level, the options apply to all files in your project. For more information about the build options, refer to your Analog Devices VisualDSP++ documentation.

The default settings for **Custom** are the same as the **Release** project configuration in VisualDSP++ software, except for the compiler options discussed in the next section “Default Project Options in Custom” on page 3-44. **Custom** uses different compiler optimization levels to preserve important features of the generated code.

### **Default Project Options in Custom**

When you create a new project or build a model to your Analog Devices processor, your project and model inherit the build configuration settings from the configuration **Custom**. The settings in **Custom** differ from the settings in the default **Debug** and **Release** configurations in VisualDSP++ software in the compiler settings.

For the compiler options, **Custom** uses the **Function(-o2)** compiler setting. The VisualDSP++ software default **Release** configuration uses **File(-o3)**, a slightly more aggressive optimization model.

For memory configuration, where **Release** uses the default memory model that specifies near functions and data, **Custom** specifies near functions and data—the **-m11** memory model—because some custom hardware might not support far data or aggregate data. Your VisualDSP++ documentation provides complete details on the compiler build options.

You can change the individual settings or the build configuration within VisualDSP++ IDE. Build configuration options that do not appear on these panes default to match the settings for the Release build configuration in VisualDSP++ software.

## Optimizing Embedded Code with Target Function Libraries

### In this section...

“About Target Function Libraries and Optimization” on page 3-46

“Using a Processor-Specific Target Function Library to Optimize Code” on page 3-48

“Process of Determining Optimization Effects Using Real-Time Profiling Capability” on page 3-49

“Reviewing Processor-Specific Target Function Library Changes in Generated Code” on page 3-50

“Reviewing Target Function Library Operators and Functions” on page 3-52

“Creating Your Own Target Function Library” on page 3-52

### About Target Function Libraries and Optimization

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

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

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

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



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

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

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

---

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

---

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

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

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

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

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

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

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

---

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

### Reviewing Code Manually

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

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

### Using Model-to-Code Tracing

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

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

The Real-Time Workshop Report window opens on your desktop. For more information about the report, refer to the Real-Time Workshop Embedded Coder documentation.

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

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

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

## Using a File Differencing Scheme

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

- 1 Generate your project with the default C89/C90 ANSI target function library. Use **Create Project**, **Archive Library**, or **Build** for the **Build action** in the Embedded IDE Link options.
- 2 Save the project to a new name—*newproject1*.
- 3 Go back to the configuration parameters for your model, and select a target function library appropriate for your processor.
- 4 Regenerate your project.
- 5 Save the project with a new name—*newproject2*

- 6 Compare the contents of the *modelName.c* files from `newproject1` and `newproject2`. The differences between the files show the target function library induced code changes.

## Reviewing Target Function Library Operators and Functions

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

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

```
RTW.viewTf1
```

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

## Creating Your Own Target Function Library

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

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

## Model Reference

In this section...
“How Model Reference Works” on page 3-53
“Using Model Reference” on page 3-54
“Configuring Targets to Use Model Reference” on page 3-56

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. For this discussion, you need to know 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 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 software detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (.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 software rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these setting through the **Model Reference** pane of the Configuration Parameters dialog box.

## **Model Reference in Code Generation**

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

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

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

## **Using Model Reference**

With few limitations or restrictions, Embedded IDE Link 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 Analog Devices targets is that you must set the **Build action** (select **Configuration Parameters > Embedded IDE Link**) 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**.
- 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 the `Archive_library` setting removes the following options from the dialog box:

- **Interrupt overrun notification method**
- **Compiler options string**
- **Linker options string**
- **System stack size (MAUs)**
- **Profile real-time execution**

### **Target Preferences Blocks in Reference Models**

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

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 software does not allow you to use the following blocks or S-functions in reference models:

- No noninlined S-functions
- None of the following blocks:
  - Custom Board (Target Preferences)
  - Memory Allocate
  - Memory Copy
  - Idle Task
  - Hardware Interrupt for SHARC, TigerSHARC, or Blackfin DSPs

### **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 targets files.
- When you generate code from a model that references other models, you must configure the top-level model and the referenced models for the same system target file.
- Real-Time Workshop software builds and Embedded IDE Link software 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 prior to MATLAB release R14SP3, use the following command to set the `ModelReferenceCompliant` flag to `On` to make your model compatible with model reference:

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

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



# Verification

---

- “What is Verification?” on page 4-2
- “Verifying Generated Code via Processor-in-the-Loop” on page 4-3
- “Profiling Code Execution in Real-Time” on page 4-9
- “System Stack Profiling” on page 4-18

## What is Verification?

Verification consists broadly of running generated code on a processor and verifying that the code does what you intend. The components of Embedded IDE Link 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 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 processor hardware
- Stack usage profiling — A tool that lets you see how your application uses the CPU stack

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

## In this section...

“What is Processor-in-the-Loop Cosimulation?” on page 4-3

“About the PIL Block” on page 4-4

“Preparing Your Model to Generate a PIL Application” on page 4-5

“Setting Model Configuration Parameters to Generate the PIL Application” on page 4-6

“Creating the PIL Block Application from a Model Subsystem” on page 4-6

“Running Your PIL Application to Perform Cosimulation and Verification” on page 4-7

“PIL Issues and Limitations” on page 4-7

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

Processor in the loop (PIL) cosimulation is a technique to help you evaluate how well an algorithm, such as a control system or signal processing algorithm, operates on the processor selected for the application.

---

**Note** PIL requires Real-Time Workshop Embedded Coder software.

---

*Cosimulation* reflects a division of labor where Simulink software models the plant or test harness, while code generated from an algorithm in the model runs on the processor hardware.

During the Real-Time Workshop Embedded Coder software code generation process, you can create a PIL block from one of several Simulink software 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.

*Definitions*

## **PIL Algorithm**

The algorithmic code, such as the signal processing algorithm, to test during the PIL cosimulation. The PIL algorithm is in compiled object form to enable verification at the object level.

## **PIL Application**

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

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

## **PIL Block**

A block you create from a subsystem in a model. When you run the simulation, the PIL block acts as the interface between the model and the PIL application running on the processor.

## **About the PIL Block**

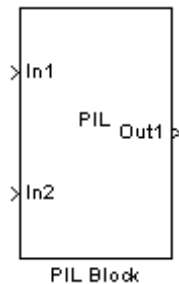
The PIL cosimulation block is the Simulink software block interface to PIL and the interface between the Simulink model and the executable PIL application running on the processor. Simulink model simulation inputs and outputs of the PIL cosimulation block match the input and output specification of the PIL algorithm.

The block is a basic building block that enables you to perform the following operations:

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



The PIL block inherits the shape and signal names from the source subsystem in your model, as shown in the following example. Inheritance is convenient for copying the PIL block into the model to replace the original subsystem for cosimulation.



## Preparing Your Model to Generate a PIL Application

PIL verification begins with a model of the process to verify. Follow these steps to prepare your model to create a PIL application and PIL block:

### 1 Develop the model of the process to simulate.

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

For information about building Simulink software 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 contains the Target Preferences block to add to your system. Configure the Target Preferences block for your processor. For more information, refer to

## Setting Model Configuration Parameters to Generate the PIL Application

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

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

Use the following steps:

- 1** 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 Embedded IDE Link software.
  - c** Under **Target selection**, set the **System target file** to .
- 2** Configure the model to perform PIL building and PIL block creation.
  - a** Select Embedded IDE Link on the **Select** tree.
  - b** On the **Build action** list, select `Create_processor_in_the_loop_project` to enable PIL.
  - c** Click **OK** to close the Configuration Parameters dialog box.

## Creating the PIL Block Application from a Model Subsystem

Using PIL and PIL blocks to verify your processes begins with a Simulink model of your process. To see an example, refer to the demo Getting Started with Application Development in the demos for Embedded IDE Link.

---

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

---

To create a PIL block, perform the following steps:

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

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

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

- 2 Copy the new PIL block from the new model to your model, either in parallel to your masked subsystem to simulate the subsystem processes concurrently, or replace your subsystem with the PIL block.

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

## Running Your PIL Application to Perform Cosimulation and Verification

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

## PIL Issues and Limitations

Consider the following issues when you work with PIL blocks.

### Generic PIL Issues

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

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

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

To use PIL, select the target file provided by Embedded IDE Link software.

## Profiling Code Execution in Real-Time

In this section...
“Overview” on page 4-9
“Profiling Execution by Tasks” on page 4-10
“Profiling Execution by Subsystems” on page 4-13

### Overview

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

Execution profiler supports profiling your code two ways:

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

---

**Note** To perform execution profiling, you must generate your project from a model in Simulink modeling environment and you must select the system target file `vdspLink_ert.tlc` in the model configuration parameters.

---

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

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

The profiler generates output in the following formats:

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

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

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

Use this general process for profiling your project:

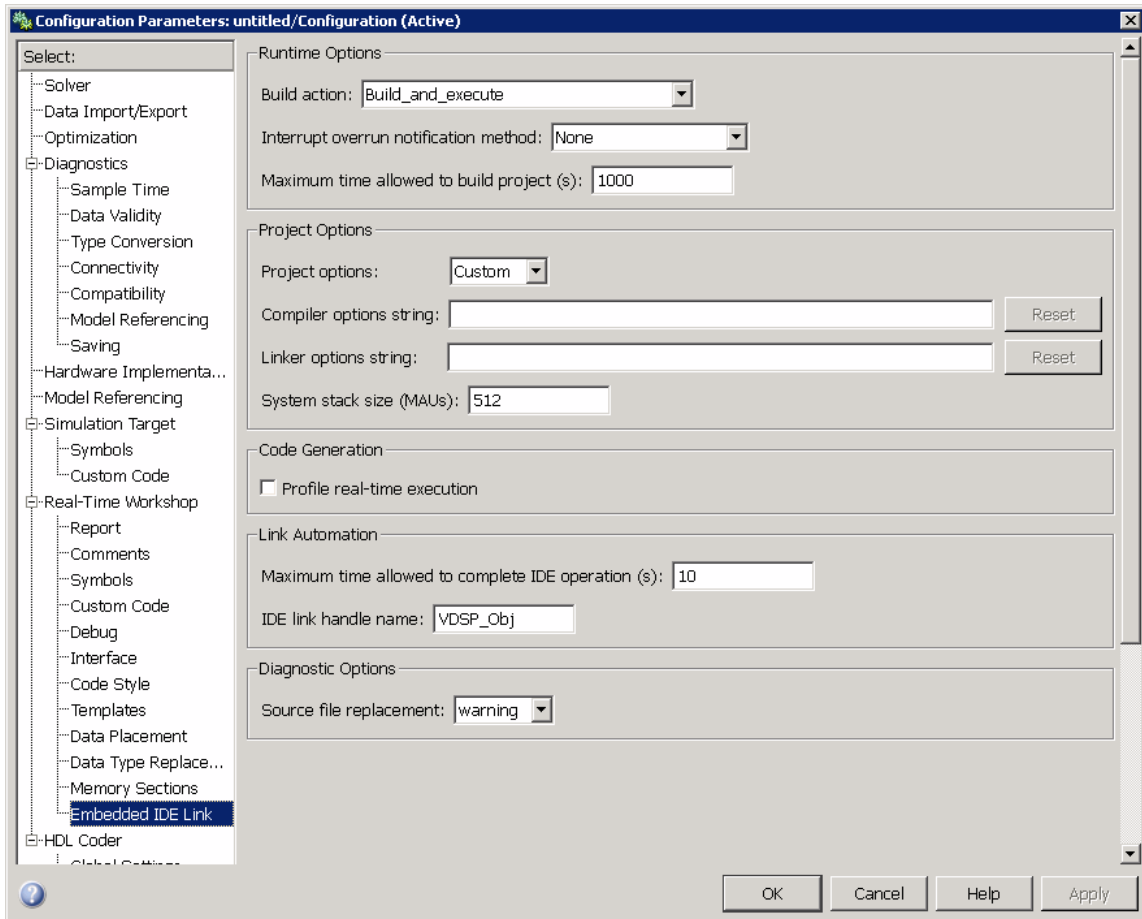
- 1** Create your model in Simulink modeling environment.
- 2** Enable execution profiling in the configuration parameters for your model.
- 3** Run your application.
- 4** Stop your application.
- 5** Get the profiling results with the `profile` function.

The following sections describe profiling your projects in more detail.

## **Profiling Execution by Tasks**

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

- 1** Open the Configuration Parameters dialog box for your model.
- 2** Select `Embedded IDE Link` from the **Select** tree. The pane appears as shown in the following figure.




**3** Select **Profile real-time execution**. The **Profile by** list appears.

**4** On the **Profile by** list, select **Task** to enable real-time task profiling.

**5** Assign a name for the object handle in **IDE link handle name**. Embedded IDE Link software exports this object to your MATLAB workspace with the name you enter.

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

To view the execution profile for your model:

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

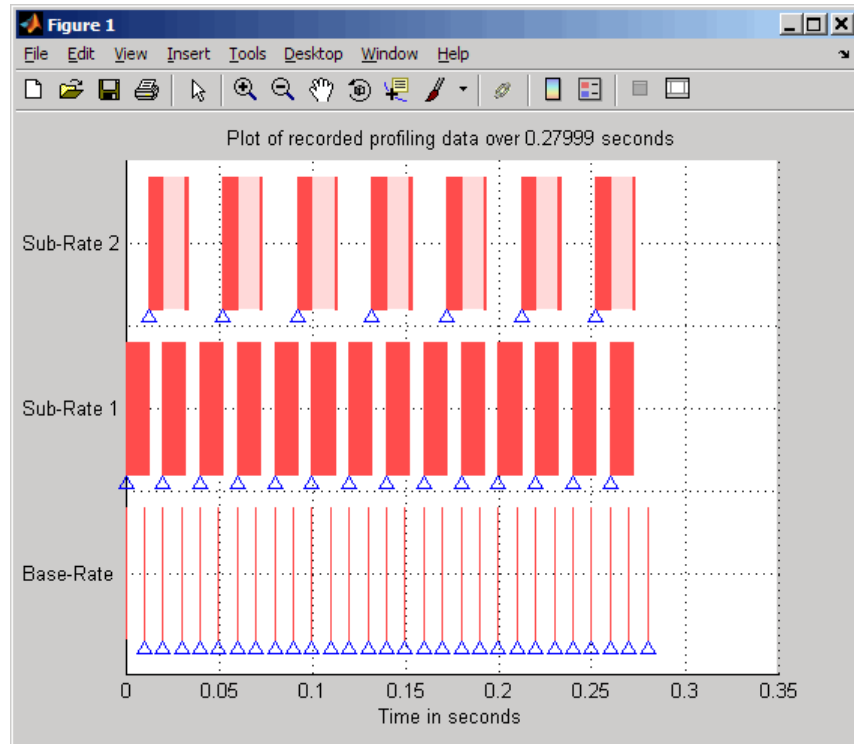
```
profile(handlename, execution , report )
```

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

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

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



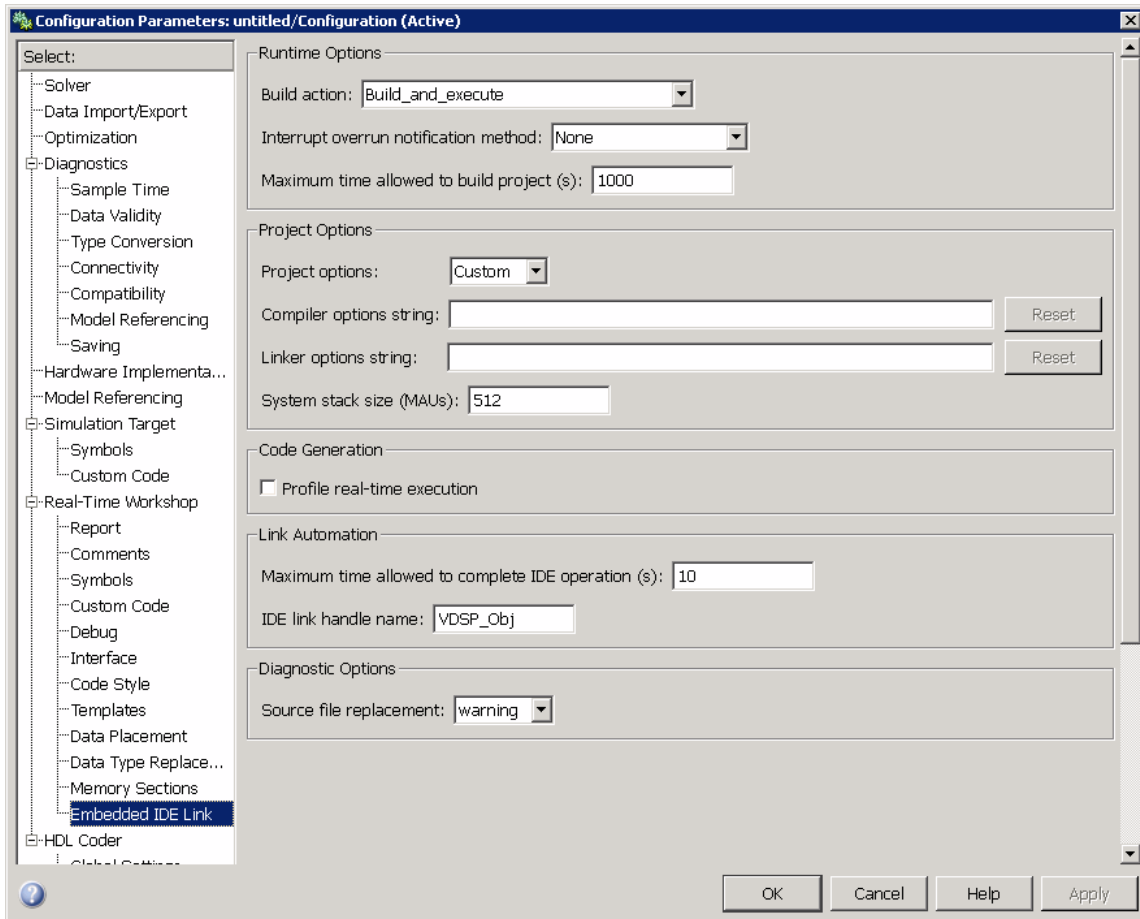


## Profiling Execution by Subsystems

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

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

- 1 Open the Configuration Parameters dialog box for your model.
- 2 Select Embedded IDE Link from the **Select** tree. The pane appears as shown in the following figure.




**3** Select **Profile real-time execution**.

**4** On the **Profile by** list, select **Atomic** subsystem to enable real-time subsystem execution profiling.

**5** Assign a name for the object handle in **IDE link handle name**. Embedded IDE Link software exports this object to your MATLAB workspace with the name you enter.

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

To view the execution profile for your model:

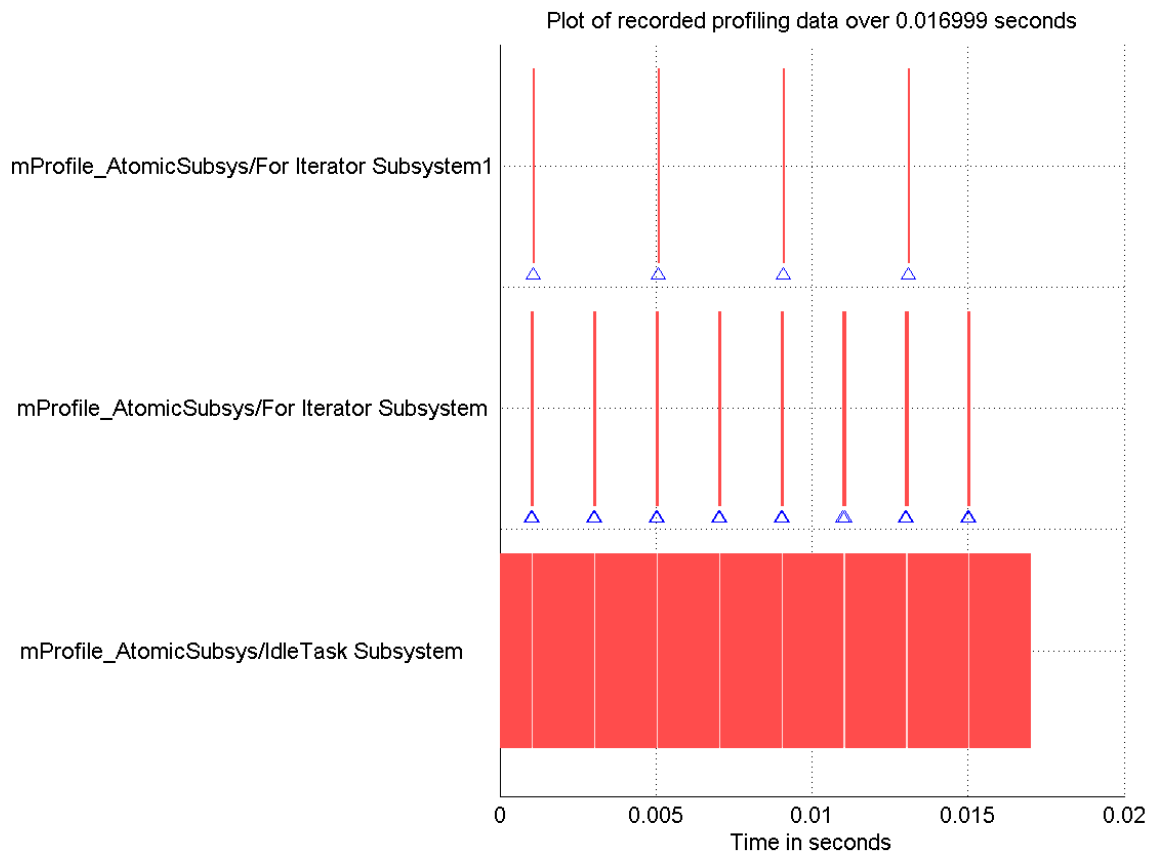
- 1** Click **Incremental build** () on the model toolbar to generate, build, load, and run your code on the processor.
- 2** To stop the running program, select **Debug > Halt** in VisualDSP IDE, or use `halt(handlename)` from the MATLAB command prompt. Gathering profile data from a running program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

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

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

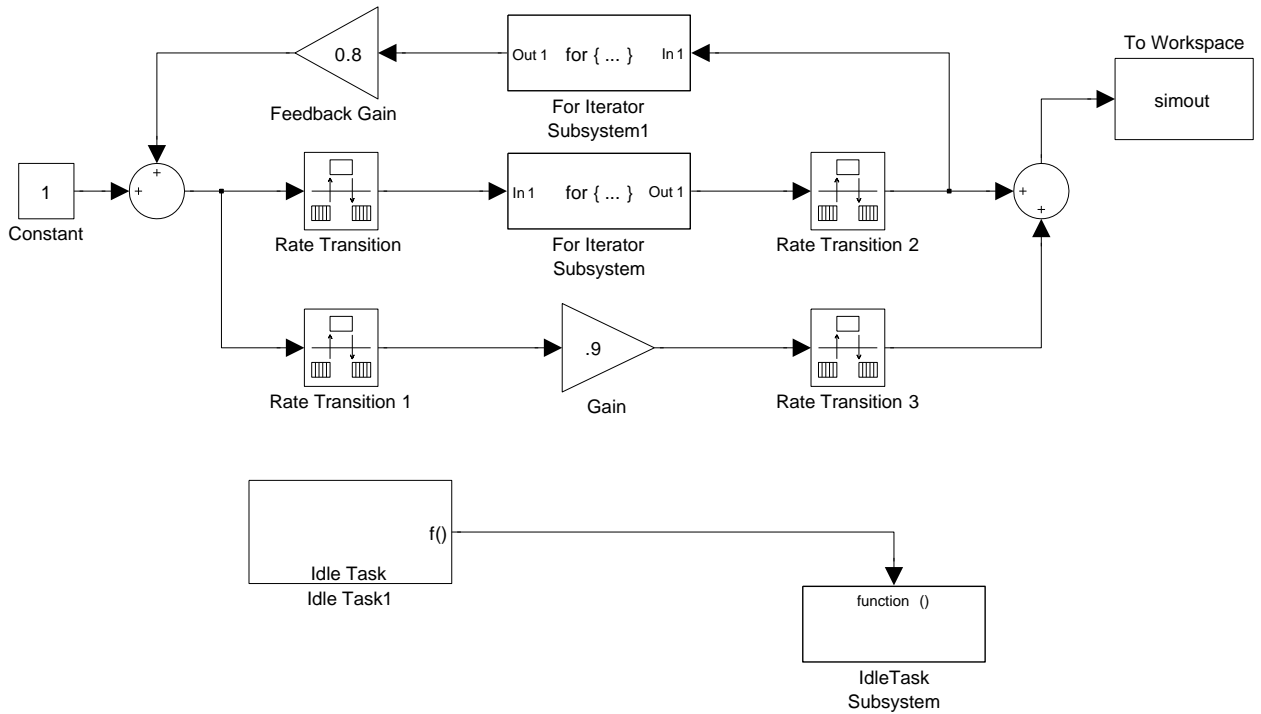
Refer to `profile` for more information.

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



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

## Atomic Subsystem Profiling



## System Stack Profiling

### In this section...

“Overview” on page 4-18

“Profiling System Stack Use” on page 4-19

### Overview

Along with real-time task execution profiling, the software enables you to determine how your application uses the processor system stack. Using the `profile` method, you can initialize and test the size and usage of the stack. This capability helps you optimize both the size of the stack and how your code uses the stack.

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

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

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

```
Maximum stack usage:
```

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

```
      name: System Stack
startAddress: [512    0]
endAddress: [1535   0]
```

```

stackSize: 1024 MAUs
growthDirection: ascending

```

The following table describes the entries in the report:

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

## Profiling System Stack Use

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

- 1 Load an application.
- 2 Set up the stack to enable profiling.

- 3** Run your application.
- 4** Request the stack profile information.

---

**Note** If your application initializes the stack with known values when you run it, stack usage is reported as 100%. The value does not correctly reflect the stack usage.

---

Follow these steps to profile the stack as your application interacts with it. In this example, `vd` is an existing `adivdsp` object.

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

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

With the **setup** input argument, `profile` writes a known pattern into the addresses that compose the stack. For all processors, the pattern is `A5`. As long as your application does not write the same pattern to the system stack, `profile` can report the stack usage correctly.

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

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

The following example demonstrates setting up and profiling the stack. The `adivdsp` object `vd` must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a SHARC ADSP-2136x simulator.

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



Maximum stack usage:

System Stack: 0/512 (0%) MAUs used.

```
        name: System Stack
startAddress: [785920      1]
endAddress: [786431      1]
stackSize: 512 MAUs
growthDirection: ascending
```

```
run(vd)
halt(vd)
profile(vd, 'stack', 'report')
```

Maximum stack usage:

System Stack: 91/512 (17.77%) MAUs used.

```
        name: System Stack
startAddress: [785920      1]
endAddress: [786431      1]
stackSize: 512 MAUs
growthDirection: ascending
```



# Function Reference

---

Constructor (p. 5-2)	adivdsp object constructor
IDE Operations (p. 5-3)	Work with VisualDSP++ IDE
Processor Operations (p. 5-4)	Control processor
Debug Operations (p. 5-5)	Perform project and code debugging
Read/Write Operations (p. 5-6)	Access memory on the processor
Get Information Operations (p. 5-7)	Project and memory operations
Object Information (p. 5-8)	Object property information
Status Operations (p. 5-9)	Determine processor operation status
Session Operations (p. 5-10)	Manage VisualDSP++ sessions
Verification (p. 5-11)	Functions that help verify code performance

## **Constructor**

adivdsp

(For VisualDSP++) Create object to session in VisualDSP++ IDE

## IDE Operations

activate	(For VisualDSP++) Make specified project, file, or configuration active
add	(For VisualDSP++) Add file or data type to active project
build	(For VisualDSP++) Build or rebuild current project
cd	(For VisualDSP++) Set IDE working directory
close	(For VisualDSP++) Close file in IDE window
dir	(For VisualDSP++) Files and directories in current IDE window
new	(For VisualDSP++) New text, project, or configuration file
open	(For VisualDSP++) Open specified file
remove	(For VisualDSP++) Remove file from active project in IDE window
save	(For VisualDSP++) Save file
visible	(For VisualDSP++) Visibility of IDE window

## Processor Operations

halt	(For VisualDSP++) Halt program execution by processor
load	(For VisualDSP++) Load file into processor
profile	(For VisualDSP++) Code execution profile and stack usage information
reset	(For VisualDSP++) Stop program execution and reset processor
run	(For VisualDSP++) Execute program loaded on processor

## Debug Operations

insert

(For VisualDSP++) Insert breakpoint  
in file

## **Read/Write Operations**

read

(For VisualDSP++) Read data from processor memory

write

(For VisualDSP++) Write data to processor memory block



## Get Information Operations

address

(For VisualDSP++) Return address and memory type of specified symbol

info

(For VisualDSP++) Property names and values for active session

## **Object Information**

display

(For VisualDSP++) Properties of  
adivdsp object

## Status Operations

`isrunning`

(For VisualDSP++) Determine whether processor is executing process

`isvisible`

(For VisualDSP++) Determine if IDE is running on desktop and window is open

## Session Operations

`listsessions`

(For VisualDSP++) List existing sessions

## Verification

profile

(For VisualDSP++) Code execution  
profile and stack usage information



# Functions — Alphabetical List

---

# activate

---

**Purpose** (For VisualDSP++) Make specified project, file, or configuration active

**Syntax**

```
activate(vd, 'my_project.dpj', 'project')
activate(vd, 'my_file', 'text')
activate(vd, 'my_config', 'buildcfg')
```

**Description** `activate(vd, 'my_project.dpj', 'project')` uses handle `vd` to activate the project named `my_project.dpj` in the IDE. If `my_project.dpj` does not exist in the IDE, MATLAB software issues an error that explains that the specified project does not exist.

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

```
activate(vd, 'try11.dpj')
```

Embedded IDE Link software cannot tell which project named `try11.dpj` 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 VisualDSP++ IDE directory to find the first project with the specified name. If the search finds the project, the software 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, the software activates the project and returns.
- 3** If Embedded IDE Link software search cannot find a project with the specified name in the VisualDSP++ IDE or on the MATLAB path, the software returns an error saying it could not find the specified project.

`activate(vd, 'my_file', 'text')` If the document window is open, this command makes the file `my_file` active in the document window. If the



document window is closed, MATLAB returns an error explaining that the document window is closed, and does not activate any file. If the specified file does not exist in the project, MATLAB returns an error. `activate` supports the text file extensions on the following list:

- `.txt`
- `.c`
- `.html`
- `.xml`

`activate(vd, 'my_config', 'buildcfg')` If the IDE application is open, and it contains an active project, `activate` makes build configuration `my_config` the active configuration in the IDE. Otherwise, MATLAB returns an error message stating that there is no active project and it could not activate the specified build configuration.

**See Also**`new``remove`

# add

---

**Purpose** (For VisualDSP++) Add file or data type to active project

**Syntax** `add(vd, 'my_file')`

**Description** `add(vd, '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 software 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(vd, 'add.txt');
```

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

```
add(vd, 'fullpathToFile\add.txt');
```

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

Supported File Type	File Extension
ANSI 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
VisualDSP++ support file	*.vdk

**See Also** `activate`  
`cd`

open

remove

# address

---

**Purpose** (For VisualDSP++) Return address and memory type of specified symbol

**Syntax** `a = address(vd, 'symbolstring')`

**Description** `a = address(vd, 'symbolstring')` returns the address and memory type values for the symbol identified by `symbolstring`. 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.

**See Also** `read`  
`write`

**Purpose** (For VisualDSP++) Create object to session in VisualDSP++ IDE

**Syntax**

```
vd = adivdsp
vd = adivdsp('propname1',propvalue1,'propname2',propvalue2,
,'timeout',value)
vd = adivdsp('my_session')
```

**Description**

vd = adivdsp opens the VisualDSP++ software for the most recent active session, if the IDE is not running. After that, it creates an object vd that references the newly-opened session. If the IDE is running, adivdsp returns object vd that connects to the active session in the IDE.

adivdsp creates an interface between MATLAB software and Analog Devices VisualDSP++ software. If this is the first time you have used adivdsp, you must supply a session name as an input argument (refer to the next syntax).

---

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

---

vd = adivdsp('sessionname','name','procnum','number',...) returns an object handle vd that you use to interact with a processor in the IDE from MATLAB.

You use the debug 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. adivdsp also enables you to create an array of objects for a multiprocessor board, where each object refers to one processor on the board. When vd is an array of objects, any method called with vd as an input argument is sent sequentially to all processors connected to the adivdsp object. VisualDSP++ software provides the communication between the IDE and the processor.

Parameters that you pass as input arguments to adivdsp 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). Although you can define any `adivdsp` object property when you create the object, there are several important properties that you must provide during object construction. These properties must be properly delineated when you create the object. The required input arguments are

- `sessionname` — Specifies the session to connect to. This session must exist in the session list. `adivdsp` does not create new sessions. The resulting object refers to a processor in `sessionname`. To see the list of sessions, use `listsessions` at the MATLAB command prompt.
- `procnum`— Specifies the processor to connect to in `sessionname`. The default value for `procnum` is 0 for the first processor on the board. If you omit the `procnum` argument, `adivdsp` connects to the first processor. `procnum` can also be an array of processor indexes on a multiprocessor board. Using an array results in the `adivdsp` object `vd` being an array of handles that correspond to the specified processors.

After you build the `adivdsp` object `vd`, you can review the object property values with `get`, but you cannot modify the `sessionname` and `procnum` property values.

To connect to the active session in IDE, omit the `sessionname` property in the syntax. If you do not pass `sessionname` as an input argument, the object defaults to the active session in the IDE.

Use `listsessions` to determine the number for the desired DSP processor(s). If your IDE session is single processor or to connect to processor zero, you can omit the `procnum` property definition. If `procnum` is not passed as an input argument, the object defaults to `procnum = 0` (zero-based).

```
vd =  
adivdsp('propname1',propvalue1,'propname2',propvalue2,  
, 'timeout', value) sets the global time-out value to value in vd.  
MATLAB waits for the specified time-out value to get a response from  
the IDE application. If the IDE does not respond within the allotted  
time-out period, MATLAB exits from the evaluation of this function.
```

`vd = adivdsp('my_session')` connects to `my_session` if the session exists in the session list and the IDE is not already running. In this case, MATLAB starts VisualDSP++ IDE for the session named `my_session`.

The following list shows some other possible cases and results of using `adivdsp` to construct an object that refers to `my_session`.

- If `my_session` does not exist in the session list and the IDE is not already running, MATLAB returns an error stating that `my_session` does not exist in the session list.
- When `my_session` is the current active session and the IDE is already running, MATLAB connects to the IDE for this session.
- If `my_session` is not the current active session, but exists in the session list, and the IDE is already running, MATLAB displays a dialog box asking if you want to switch to `my_session`. If you choose to switch to `my_session`, all existing handles you have to other sessions in the IDE become invalid. To connect to the other sessions you need to use `adivdsp` to recreate the objects for those sessions.
- If `my_session` does not exist in the session list and the IDE is already running, MATLAB returns an error, explaining that the session `my_session` does not exist in the session list.

## Examples

These examples demonstrate some of the operation of `adivdsp`.

```
vd = adivdsp('sessionname','my_session','procnum',0);
```

returns a handle to the first DSP processor for session `my_session`.

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

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

# adivdsp

---

`vd = adivdsp` without input arguments constructs the object `vd` with the default property values, returning a handle to the first DSP processor for the active session in the IDE.

`vd = adivdsp('sessionname', 'my_session');` returns a handle to the first DSP processor for the session `my_session`.

## See Also

`listsessions`



---

<b>Purpose</b>	(For VisualDSP++) Build or rebuild current project
<b>Syntax</b>	<code>build(vd)</code> <code>build(vd,timeout)</code> <code>build(vd,'all')</code> <code>build(vd,'all',timeout)</code>
<b>Description</b>	<p><code>build(vd)</code> incrementally builds the active project. Incremental builds recompile only source files in your project that you changed or added after the most recent build. <code>build</code> uses the file time stamp to determine whether to recompile a file. After recompiling the source files, <code>build</code> links the files to make a new program file.</p> <p><code>build(vd,timeout)</code> incrementally builds the active project with a time limit for how long MATLAB waits for the build process to complete. <code>timeout</code> defines the upper limit in seconds for the period the <code>build</code> routine waits for confirmation that the build process is finished. If the build process exceeds the time-out period, control returns to MATLAB immediately with a time-out error. Usually, <code>build</code> causes the processor to initiate a restart, even if it reaches the time-out limit. The time-out error in MATLAB indicates that confirmation was not received before the time-out period expired. The build action continues. Generally, the build and link process finishes successfully in spite of the time-out error.</p> <p><code>build(vd,'all')</code> rebuilds all the files in the active project.</p> <p><code>build(vd,'all',timeout)</code> rebuilds all the files in the active project applying the time-out limit on how long MATLAB waits for the build process to complete.</p>
<b>See Also</b>	<code>isrunning</code> <code>open</code>

# cd

---

**Purpose** (For VisualDSP++) Set IDE working directory

**Syntax**  
`wd = cd(vd)`  
`cd (vd, 'directory')`

**Description** `wd = cd(vd)` returns the current IDE working directory, where `vd` is an `adivdsp` object that refers to the VisualDSP++ window, or a vector of objects.

`cd (vd, '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** (For VisualDSP++) Close file in IDE window

---

**Note** `close( , text )` produces an error.

---

**Syntax** `close(vd, 'filename', 'filetype')`

**Description** `close(vd, 'filename', 'filetype')` closes the file named 'filename' in the active project in the vd 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 changes you made after the most-recent save operation. Use `save` to preserve your changes before you close the file.

---

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, `vd` is an `adivdsp` object handle to the IDE.

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

`close(vd, 'my.dpj', 'project')` — Closes the open project `my.dpj`

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

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

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

# close

---

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

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

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

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

## See Also

`add`

`open`

`save`

**Purpose** (For VisualDSP++) Files and directories in current IDE window

**Syntax** `dir(vd)`  
`d = dir(vd)`

**Description** `dir(vd)` lists the files and directories in the IDE working directory, where `vd` is the handle to the IDE. `vd` can be either a single handle, or a vector of handles. When `vd` is a vector, `dir` returns the files and directories for each handle.

`d = dir(vd)` 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
<code>name</code>	Name of the file or directory.
<code>date</code>	Date of most recent file or directory modification.
<code>bytes</code>	Size of the file in bytes. Directories return 0 for the number of bytes.
<code>isdirectory</code>	0 if 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** (For VisualDSP++) Properties of adivdsp object

**Syntax** `display(vd)`

**Description** `display(vd)` displays the properties and property values of the adivdsp object `vd`.

For example, when you create `vd` associated with the session `Testsession` and the processor is the ADSP-BF533, `display(vd)` returns the following information in the MATLAB command window:

```
display(vd)
```

```
ADIVDSP Object:
  Session name      : Testsession
  Processor name    : ADSP-BF533
  Processor type    : ADSP-BF533
  Processor number  : 0
  Default timeout   : 10.00 secs
```

**See Also** `get` in the MATLAB Function Reference

**Purpose**

(For VisualDSP++) Halt program execution by processor

**Syntax**

```
halt(vd)
halt(vd,timeout)
```

**Description**

`halt(vd)` 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 time-out period defaults to the global time-out period specified in `vd`. Use `get(vd, 'timeout')` to determine the global time-out 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(vd, 'pc')` function can determine the memory address where the processor stopped after you use `halt`.

`halt(vd,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 time-out period, the routine returns with a time-out error.

**Examples**

Use one of the provided demonstration programs to show how `halt` works. From the VisualDSP++ demonstration programs, load and run one of the demonstration projects.

At the MATLAB prompt, create an object that refers to a VisualDSP++ IDE session.

```
vd = adivdsp
```

Check whether the program is running on the processor.

# halt

---

```
isrunning(vd)

ans =

    1

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

ans =

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

ans =

    0
```

Issuing the halt stops the process on the processor. Checking in VisualDSP++ IDE confirms that the process has stopped.

## See Also

```
isrunning
reset
run
```



**Purpose** (For VisualDSP++) Property names and values for active session

**Syntax** `objinfo = info(vd)`

**Description** `objinfo = info(vd)` returns the property names and values associated with the active session and the processors for that session. The following table shows the properties `info` returns for `vd` and provides brief descriptions of the properties.

Property Name	Data Type	Description
procname	String	Provides the name of the processor.
proctype	String	Provide the platform name the session uses.
revision	String	Reports the silicon revision of the processor. Does not apply to simulators.

**Examples** When you have an `adivdsp` object `vd`, `info` provides information about the object.

```
vd = adivdsp('sessionname', 'Testsession')
```

```
ADIVDSP Object:
```

```
Session name      : Testsession
Processor name    : ADSP-BF533
Processor type    : ADSP-BF533
Processor number  : 0
Default timeout   : 10.00 secs
```

```
objinfo = info(vd)
```

```
objinfo =
```

```
procname: 'ADSP-BF533'
proctype: 'ADSP-BF533'
```

revision: ''

objinfo.procname

ans =

ADSP-BF533

## See Also

display

adivdsp

**Purpose** (For VisualDSP++) Insert breakpoint in file

**Syntax**  
`insert(vd,addr)`  
`insert(vd,'filename','linenumber')`

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

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

**See Also**  
address  
run

# isrunning

---

**Purpose** (For VisualDSP++) Determine whether processor is executing process

**Syntax** `isrunning(vd)`

**Description** `isrunning(vd)` 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.

```
vd = adivdsp
```

```
ADIVDSP Object:
```

```
Session name      : Testsession
```

```
Processor name    : ADSP-BF533
```

```
Processor type    : ADSP-BF533
```

```
Processor number  : 0
```

```
Default timeout   : 10.00 secs
```

```
visible(vd,1)
```

```
load(vd, 'adi.dxe', 'program')
```

```
run(vd)
```

```
isrunning(vd)
```

```
ans =
```

```
1
```

```
halt(vd)
```

```
isrunning(vd)
```

```
ans =
```

```
0
```

## See Also

halt

load

run

# isvisible

---

<b>Purpose</b>	(For VisualDSP++) Determine if IDE is running on desktop and window is open
<b>Syntax</b>	<code>isvisible(vd)</code>
<b>Description</b>	<code>isvisible(vd)</code> determines whether the VisualDSP++ IDE is running on the desktop and the window is open. <code>isvisible</code> returns either 1 indicating that the IDE is running and the window is open, or 0 indicating that either the IDE is running in the background or is not running.
<b>See Also</b>	<code>visible</code>

**Purpose** (For VisualDSP++) List existing sessions

**Syntax**  
`list = listsessions`  
`list = listsessions('verbose')`

**Description** `list = listsessions` returns `list` that contains a listing of all of the sessions by name currently in the development environment.

`list = listsessions('verbose')` adds the optional input argument `verbose`. When you include the `verbose` argument, `listsessions` returns a cell array that contains one row for each existing session. Each row has three columns — processor type, platform name, and processor name.

**See Also** `adivdsp`

# load

---

**Purpose** (For VisualDSP++) Load file into processor

**Syntax** `load(vd, 'filename', timeout)`  
`load( , timeout)`

**Description** `load(vd, '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 VisualDSP++ IDE. Use the function `cd` to check or modify the VisualDSP++ working directory. Use this function only with program files that you created by a VisualDSP++ build process. When you issue the `load` command, the command waits for the period defined by `timeout` in `vd` 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**

(For VisualDSP++) New text, project, or configuration file

---

**Note** `new( , 'text' )` produces an error.

---

**Syntax**

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

**Description**

`new(vd, '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 file or 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 VisualDSP++ working directory.

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

Type String	Description
<b>text</b>	Create an empty text file in the current session.
<b>project</b>	Create a new executable project in the current session. Sometimes this is called a <i>DSP executable file</i> .
<b>projlib</b>	Create a new library project in the current session.
<b>buildcfg</b>	Create a build configuration in the active project.

**Examples**

`new(vd, 'my_project.dpj', 'project', project_type)` creates a new project 'my\_project.dpj' of type `project_type`. The `project_type` argument is optional; the default project type is DSP executable file.

`new(vd, 'my_config', 'buildcfg')` creates a new build configuration, named `my_config`, in the active project.

## new

---

### **See Also**

activate

close

save

**Purpose** (For VisualDSP++) Open specified file

---

**Note** `open( , 'text' )` produces an error.

---

**Syntax**

```
open(vd, 'filename')
open( , 'filetype')
```

**Description** `open(vd, '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 text file. The following table presents the possible file types and extensions.

File Type	Extension	Description
<b>text</b>	txt, c, asm, cpp, h, and all file extensions not listed elsewhere in this table	Text file
<b>project</b>	dpj	VisualDSP++ IDE project
<b>projectgroup</b>	dpg	Project group in VisualDSP++ IDE project
<b>workspace</b>	None	Workspace in VisualDSP++ IDE 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

---

`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** (For VisualDSP++) Code execution profile and stack usage information

**Syntax**  
`profile(vd, execution , 'report')`  
`profile(vd, 'stack', action)`

**Description** `profile(vd, execution , 'report')` returns execution profile measurements from the generated code. The **report** input argument is required. When you select **Profile real-time execution** in the model configuration parameters, and then build and run your model on a processor, `profile` accesses the report of the process execution.

---

**Note** Real-time task execution profiling works with hardware only. Simulators do not support the profiling feature.

---

To use `profile` to assess how your program executes in real-time, complete the following tasks with a Simulink model:

- 1** Enable real-time execution profiling in the configuration parameters and build your model.
- 2** Select whether to profile by task or subsystem.
- 3** Build your model.
- 4** Download your program to the processor.
- 5** Run the program on the processor.
- 6** Stop the running program.
- 7** Use `profile` at the MATLAB command prompt to access the profiling reports.

The HTML report contains the sections described in the following table.

Section Heading	Description
Worst case task turnaround times	Maximum task turnaround time for each task since model execution started.
Maximum number of concurrent overruns for each task	Maximum number of concurrent task overruns since model execution started.
Analysis of profiling data recorded over <i>nnn</i> seconds.	Profiling data was recorded over <i>nnn</i> seconds. The recorded data for task turnaround times and task execution times is presented in the table below this heading.

*Task turnaround time* is the elapsed time between starting and finishing the task. If the task is not preempted, task turnaround time equals the task execution time.

*Task execution time* is the time between task start and finish when the task is actually running. It does not include time during which the task may have been preempted by another task.

---

**Note** Task execution time cannot be measured directly. Task profiling infers the execution time from the task start and finish times, and the intervening periods during which the task was preempted by another task.

---

The execution time calculations do not account for processor time consumed by the scheduler while switching tasks. In cases where preemption occurs, the reported task execution times overestimate the true task execution time.

*Task overruns* occur when a timer task does not complete before the same task is scheduled to run again. Depending on how you configure the real-time scheduler, a task overrun may be handled as a real-time failure. Alternatively, you might allow a small number of task overruns to accommodate cases where a task occasionally takes longer than

normal to complete. If a task overrun occurs, and the same task is scheduled to run again before the first overrun has been cleared, concurrent task overruns are said to have occurred.

`profile(vd, 'stack', action)` returns the CPU stack usage from your application. `action` defines the stack profiling operation and accepts one of the strings in the following table:

action String	Description
<b>setup</b>	Initializes the CPU stack with a known pattern—0xA5 on all processors.
<b>report</b>	Returns the report of the stack usage from running your application.

You cannot assign the stack profile report to an output variable. The MATLAB structure output from profiling the system stack has the elements described in the following table.

Report Entry	Units	Description
System Stack	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
name	String for the stack name	Lists the name assigned to the stack.
startAddress	Decimal address and page	Lists the address of the stack start and the memory page.
endAddress	Decimal address and page	Lists the address of the end of the stack and the memory page.

# profile

---

Report Entry	Units	Description
stackSize	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
growthDirection	Not applicable	Reports whether the stack grows from the lower address to the higher address (ascending) or from higher to lower (descending).

To use `profile` to assess how your program uses the stack, complete the following tasks with a Simulink model or manually written code:

- 1** Build your model with real-time execution profiling enabled in the configuration parameters. Skip this step for custom code.
- 2** Download your program to the processor.
- 3** Run the program on the processor.
- 4** Stop the running program.
- 5** Use `profile` at the MATLAB command prompt to access the profiling reports.

You cannot assign the stack profile report to an output variable. For more information about using stack profiling, refer to “System Stack Profiling” on page 4-18.

## See Also

`load`  
`run`



**Purpose**

(For VisualDSP++) Read data from processor memory

**Syntax**

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

**Description**

`mem = read(vd, address)` returns a block of data values from the memory space of the DSP processor referenced by `vd`. 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 use a numeric vector representation of the address (see below) to define the memory type value explicitly .

Alternatively, the `vd` object has a default memory type value that is applied when 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 `vd` 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; myaddress1{2} =  
'Program(PM) Memory';  
myaddress2 myaddress2{1} = '20000'; myaddress2{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:

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

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

`mem = 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 `vd` 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 `vd` `memorytype` property value to zero. Blackfin and SHARC processors 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 <code>memorytype</code>	Numeric Entry for <code>memorytype</code>	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 control to the MATLAB command prompt. 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(vd,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(vd,'20000','int32',100)
plot(double(data))
```

## See Also

`write`

## remove

---

**Purpose** (For VisualDSP++) Remove file from active project in IDE window

**Syntax** `remove(vd, 'filename', 'filetype')`

**Description** `remove(vd, 'filename', 'filetype')` removes the file named `filename` from the active project in the `vd` 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`

**Purpose** (For VisualDSP++) Stop program execution and reset processor

**Syntax** `reset(vd,timeout)`

**Description** `reset(vd,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`

# run

---

**Purpose** (For VisualDSP++) Execute program loaded on processor

**Syntax**  
`run(vd)`  
`run(vd, 'runopt')`  
`run(..., timeout)`

**Description** `run(vd)` 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 `vd` references more than one processor, each processor calls `run` in sequence.

`run(vd, 'runopt')` includes the parameter `runopt` that defines the action of the `run` method. The options for `runopt` are listed in the following table.

<b>runopt String</b>	<b>Description</b>
<code>run</code>	Executes the run and waits to confirm that the processor is running, and then returns to MATLAB.
<code>runtohalt</code>	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 VisualDSP++ IDE, 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 time-out 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 time-out is reached. The time-out



indicates that the confirmation was not received before the time-out period elapsed.

**See Also**

halt

load

reset

# save

---

**Purpose** (For VisualDSP++) Save file

---

**Note** `save( , 'text')` produces an error.

---

**Syntax**

```
save(vd, 'filename')  
save(vd, 'filename', 'filetype')
```

**Description** `save(vd, 'filename')` saves the file named `filename` in VisualDSP++ IDE. `filename` must match the name of the file to save and you must include the file extension. You can save only open files. If you specify the `filename` parameter as `all`, every open file of the defined type is saved (refer to the `filetype` parameter in the next syntax). A null input, `[]`, for `filename` or no file name saves the current active file (the file that has focus).

`save(vd, 'filename', 'filetype')` saves the specified file in VisualDSP++ IDE.

`filetype` defines the type of file to save. In the following table you see examples of `save` that use the allowed file type definitions—`project`, `projectgroup`, and `text`.

<b>save Command</b>	<b>Description of save Operation</b>
<code>save(vd, 'all', 'project')</code>	Saves all project files
<code>save(vd, 'my.dpj', 'project')</code>	Saves the project <code>my.dpj</code>
<code>save(vd, [], 'project')</code>	Saves the active project
<code>save(vd, 'all', 'projectgroup')</code>	Saves all project files in the project groups
<code>save(vd, 'myg.dpg', 'projectgroup')</code>	Saves the project group <code>myg.dpg</code>

<b>save Command</b>	<b>Description of save Operation</b>
<code>save(vd,[],'projectgroup')</code>	Saves the projects in the active project group
<code>save(vd,'all','text')</code>	Save all text files
<code>save(vd,'text.c','text')</code>	Saves the text file <code>text.c</code>
<code>save(vd,[],'text')</code>	Save the active text file

**See Also**

adivdsp

close

load

# visible

---

**Purpose** (For VisualDSP++) Visibility of IDE window

**Syntax** `visible(vd,state)`

**Description** `visible(vd,state)` sets the visibility state of the IDE window defined by `vd`. 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 directly 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 an `adivdsp` object, the IDE visibility is set to 0 and the IDE is not visible.

**See Also** `info`  
`isvisible`

**Purpose**

(For VisualDSP++) Write data to processor memory block

**Syntax**

```
mem = write(vd, address, data)
mem = write(..., datatype)
mem = write(..., memorytype)
mem = write(..., timeout)
```

**Description**

`mem = write(vd, address, data)` writes `data`, a collection of values, to the memory space of the DSP processor referenced by `vd`. 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 `vd` 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 `vd` 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.

# write

(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:

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

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

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

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

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

`mem = write(...,datatype)` where the 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:

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

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

`mem = write(...,memorytype)` adds an optional input argument `memorytype`. Object `vd` 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 `vd` `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.

<b>String Entry for memorytype</b>	<b>Numeric Entry for memorytype</b>	<b>Processor Support</b>
'program(pm) memory'	0	Blackfin and SHARC
'data(dm) memory'	1	SHARC

# write

String Entry for memorytype	Numeric Entry for memorytype	Processor Support
'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 displays an error and returns control to the command prompt. 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(vd,[131072 1],int16([1:100]));
```

Now write a single-precision IEEE floating point value (32-bits) at address 2000A(Hex).

```
write(vd,'2000A',single(23.5));
```

For the third example, write a 2-D array of integers in row-major format (standard ANSI C code programming format) at address 131072 (decimal).

```
mlarr = int32([1:10;101:110]);  
write(vd,131072,mlarr);
```

## See Also

`hex2dec` in the MATLAB Function Reference  
`read`



# Block Reference

---

Block Library: idelinklib\_aidvds  
(p. 7-2)

Blocks for Analog Devices  
VisualDSP++

Block Library: idelinklib\_common  
(p. 7-3)

Blocks for Embedded IDE Link

## **Block Library: idelinklib\_aidvsp**

Blackfin Hardware Interrupt

SHARC Hardware Interrupt

TigerSHARC Hardware Interrupt

Generate Interrupt Service Routine

Generate Interrupt Service Routine

Generate Interrupt Service Routine

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

Idle Task

Memory Allocate

Memory Copy

Create free-running task

Allocate memory section

Copy to and from memory section



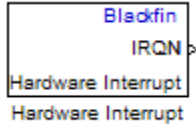
# Blocks — Alphabetical List

---

# Blackfin Hardware Interrupt

**Purpose** Generate Interrupt Service Routine

**Library** Block Library: idelinklib\_avidvsp



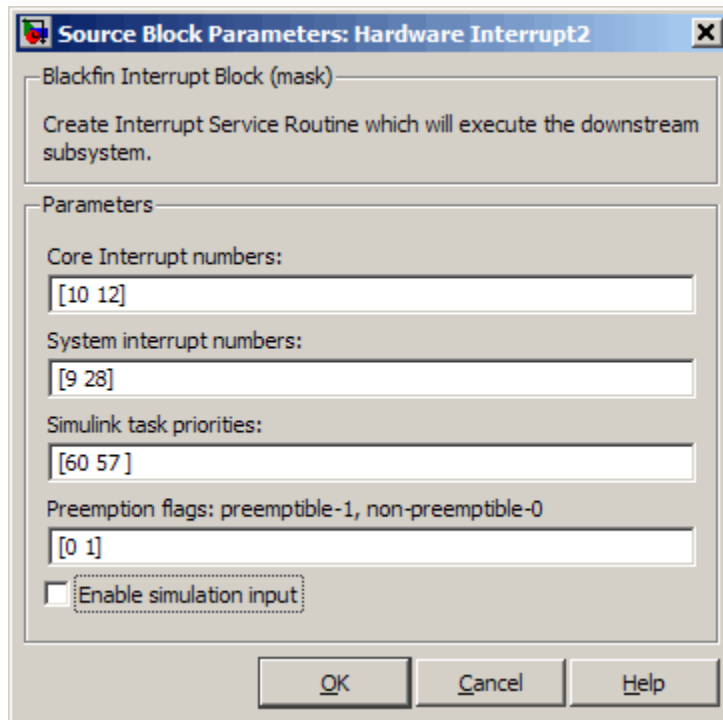
**Description** Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from the this block or an Idle Task block connected to this block. 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 Embedded IDE Link Software
Core interrupt numbers	7 to 14
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 14, where 7 through 13 are hardware driven, and 14 is software driven. Core interrupts numbered 0 to 6 are reserved and cannot be entered in this field. Each interrupt value must be unique.

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.

# Blackfin Hardware Interrupt

---

## System interrupt numbers

System interrupt numbers identify system interrupts to map to core interrupts. Enter one or more values as a vector. Each interrupt value must be unique. The valid range is generally 0 through 31, although the range depends on your processor. Some processors do not support the full range of 32 system interrupts. Embedded IDE Link software does not test for valid system interrupt values. You must verify that your values are valid for your processor. To use asynchronous scheduling, you must specify a value for at least one system interrupt number.

The block maps the first interrupt value in this field to the first core interrupt value 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. You must enter the same number of system interrupts as core interrupts.

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 model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Proper code generation requires rate transition code (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 preemptible – 1, non-preemptible – 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 software 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.

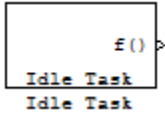
# Idle Task

---

**Purpose** Create free-running task

**Library** Block Library: idelinklib\_common

**Description**



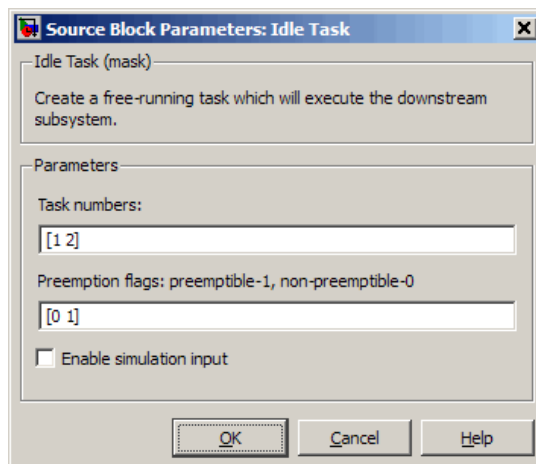
The Idle Task block, and the subsystem connected to it, specify one or more functions to execute as background tasks. All tasks executed through the Idle Task block are of the lowest priority, lower than that of the base rate task.

**Vectorized Output**

The block output comprises a set of vectors—the task numbers vector and the preemption flag or flags vector. Any preemption-flag vector must be the same length as the number of tasks vector unless the preemption flag vector has only one element. The value of the preemption flag determines whether a given interrupt (and task) is preemptible. Preemption overrides prioritization. A lower-priority nonpreemptible task can preempt a higher-priority preemptible task.

When the preemption flag vector has one element, that element value applies to all functions in the downstream subsystem as defined by the task numbers in the task number vector. If the preemption flag vector has the same number of elements as the task number vector, each task defined in the task number vector has a preemption status defined by the value of the corresponding element in the preemption flag vector.

## Dialog Box



### Task numbers

Identifies the created tasks by number. Enter as many tasks as you need by entering a vector of integers. The default values are [ 1 , 2 ] to indicate that the downstream subsystem has two functions.

The values you enter determine the execution order of the functions in the downstream subsystem, while the number of values you enter corresponds to the number of functions in the downstream subsystem.

Enter a vector containing the same number of elements as the number of functions in the downstream subsystem. This vector can contain no more than 16 elements, and the values must be from 0 to 15 inclusive.

The value of the first element in the vector determines the order in which the first function in the subsystem is executed, the value of the second element determines the order in which the second function in the subsystem is executed, and so on.

# Idle Task

---

For example, entering [2,3,1] in this field indicates that there are three functions to be executed, and that the third function is executed first, the first function is executed second, and the second function is executed third. After all functions are executed, the Idle Task block cycles back and repeats the execution of the functions in the same order.

## Preemption flags

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Task numbers** contains more than one task, you can assign different preemption flags to each task by entering a vector of flag values, corresponding to the order of the tasks in **Task numbers**. If **Task numbers** contains more than one task, and you enter only one flag value here, that status applies to all tasks.

In the default settings [0 1], the task with priority 1 in **Task numbers** is not preemptible, and the priority 2 task can be preempted.

## Enable simulation input

When you select this option, Simulink software adds an input port to the Idle Task block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

---

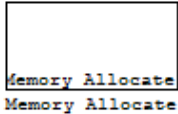
**Note** Select this check box to test asynchronous interrupt processing behavior in Simulink software.

---

**Purpose** Allocate memory section

**Library** Block Library: idelinklib\_common

## Description



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

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

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

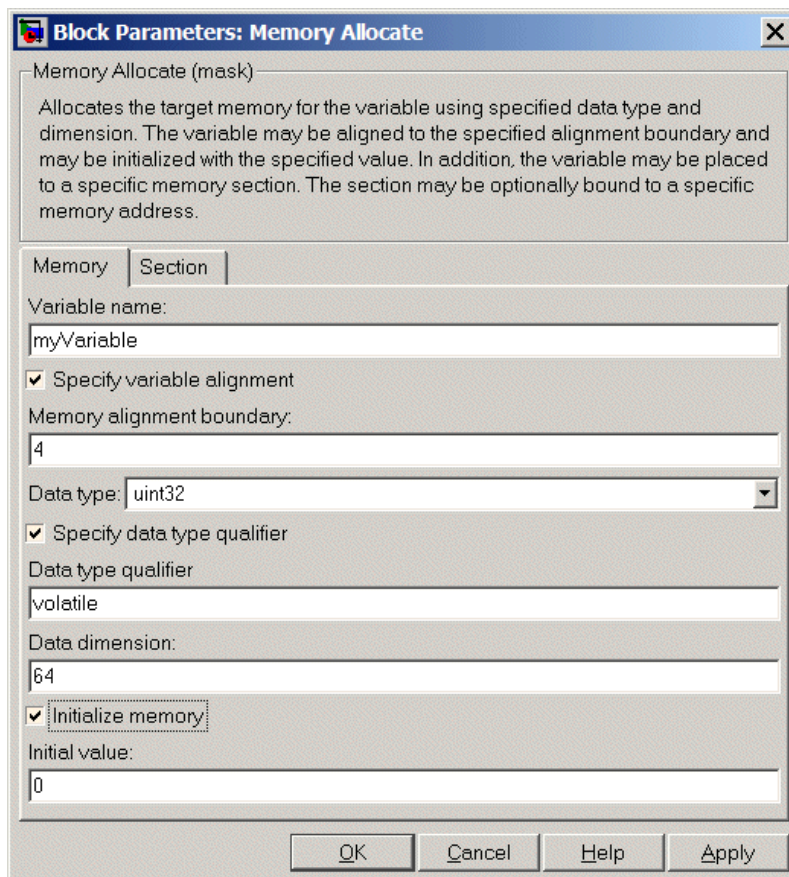
## Dialog Box

The block dialog box comprises multiple tabs:

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

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

# Memory Allocate



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

## Memory Parameters

**Block Parameters: Memory Allocate**

Memory Allocate (mask)

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

Memory | Section

Variable name:  
myVariable

Specify variable alignment

Memory alignment boundary:  
4

Data type: uint32

Specify data type qualifier

Data type qualifier  
volatile

Data dimension:  
64

Initialize memory

Initial value:  
0

OK Cancel Help Apply

You find the following memory parameters on this tab.

### Variable name

Specify the name of the variable to allocate. The variable is allocated in the generated code.

# Memory Allocate

---

## **Specify variable alignment**

Select this option to direct the compiler to align the variable in **Variable name** to an alignment boundary. When you select this option, the **Memory alignment boundary** parameter appears so you can specify the alignment. Use this parameter and **Memory alignment boundary** when your processor requires this feature.

## **Memory alignment boundary**

After you select **Specify variable alignment**, this option enables you to specify the alignment boundary in bytes. If your variable contains more than one value, such as a vector or an array, the elements are aligned according to rules applied by the compiler.

## **Data type**

Defines the data type for the variable. Select from the list of types available.

## **Specify data type qualifier**

Selecting this enables **Data type qualifier** so you can specify the qualifier to apply to your variable.

## **Data type qualifier**

After you select **Specify data type qualifier**, you enter the desired qualifier here. `volatile` is the default qualifier. Enter the qualifier you need as text. Common qualifiers are `static` and `register`. The block does not check for valid qualifiers.

## **Data dimension**

Specifies the number of elements of the type you specify in **Data type**. Enter an integer here for the number of elements.

## **Initialize memory**

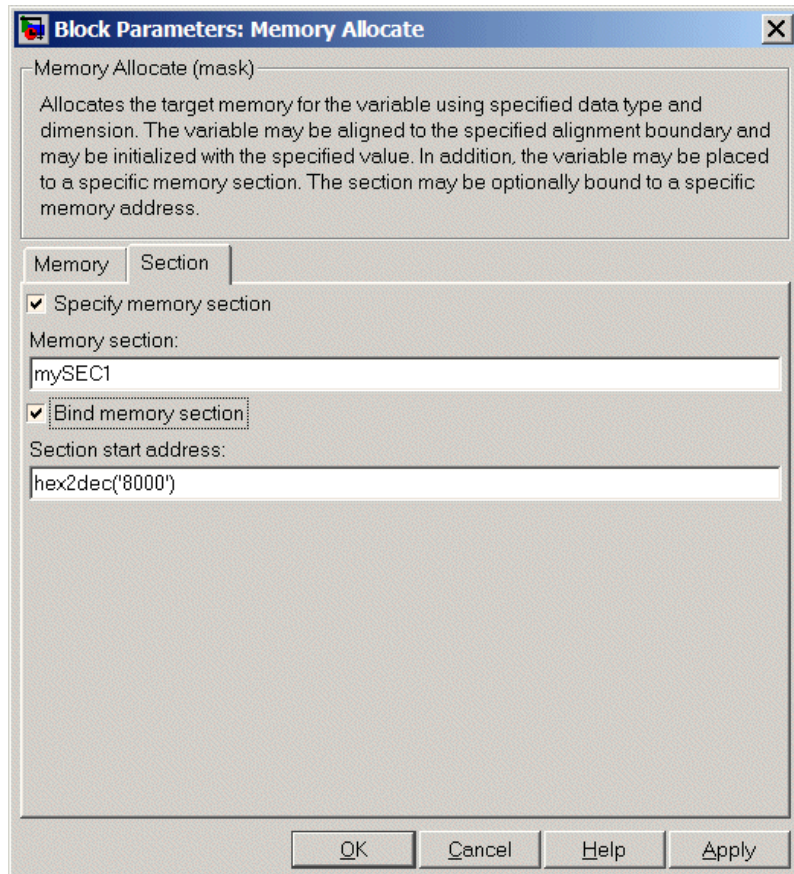
Directs the block to initialize the memory location to a fixed value before processing.

## **Initial value**

Specifies the initialization value for the variable. At run time, the block sets the memory location to this value.



## Section Parameters



Parameters on this pane specify the section in memory to store the variable.

### Specify memory section

Selecting this parameter enables you to specify the memory section to allocate space for the variable. Enter either one of the

# Memory Allocate

---

standard memory sections or a custom section that you declare elsewhere in your code.

## Memory section

Identify a specific memory section to allocate the variable in **Variable name**. Verify that the section has sufficient space to store your variable. After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use **Bind memory section** to bind the memory section to a location.

## Bind memory section

After you specify a memory section by selecting **Specify memory section** and entering the section name in **Memory section**, use this parameter to bind the memory section to the location in memory specified in **Section start address**. When you select this, you enable the **Section start address** parameter.

The new memory section specified in **Memory section** is defined when you check this parameter.

---

**Note** Do not use **Bind memory section** for existing memory sections.

---

## Section start address

Specify the address to which to bind the memory section. Enter the address in decimal form or in hexadecimal with a conversion to decimal as shown by the default value `hex2dec('8000')`. The block does not verify the address—verify that the address exists and can contain the memory section you entered in **Memory section**.

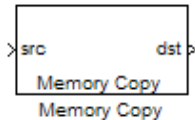
## See Also

Memory Copy

**Purpose** Copy to and from memory section

**Library** Block Library: idelinklib\_common

## Description



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

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

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

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

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

## Block Operations

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. With the options for setting memory initialization and termination, you

# Memory Copy

---

control when and how the block initializes memory, copies to and from memory, and terminates memory operations. The parameters enable you to turn on and off memory operations in all three periods independently.

Used in combination with the Memory Allocate block, this block supports building custom device drivers, such as PCI bus drivers or codec-style drivers, by letting you manipulate and allocate memory. This block does not require the Memory Allocate block to be in the model.

In a simulation, this block does not perform any operation. The block output is not defined.

## Copy Memory

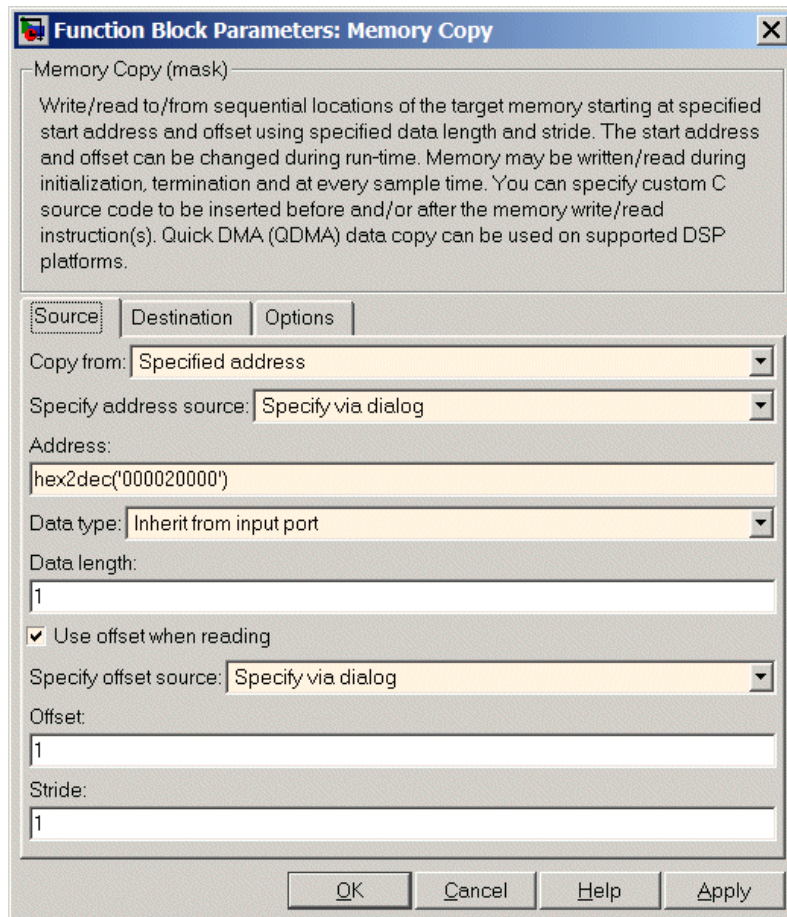
When you employ this block to copy an individual data element from the source to the destination, the block copies the element from the source in the source data type, and then casts the data element to the destination data type as provided in the block parameters.

## Dialog Box

The block dialog box contains multiple tabs:

- **Source** — Identifies the sequential memory location to copy from. Specify the data type, size, and other attributes of the source variable.
- **Destination** — Specify the memory location to copy the source to. Here you also specify the attributes of the destination.
- **Options** — Select various parameters to control the copy process.

The dialog box images show many of the available parameters enabled. Some parameters shown do not appear until you select one or more other parameters. Some parameters are not shown in the figures, but the text describes them and how to make them available.



Sections that follow describe the parameters on each tab in the dialog box.

# Memory Copy

## Source Parameters

Function Block Parameters: Memory Copy

Memory Copy (mask)

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

Source | Destination | Options

Copy from: Specified address

Specify address source: Specify via dialog

Address:  
hex2dec('000020000')

Data type: Inherit from input port

Data length:  
1

Use offset when reading

Specify offset source: Specify via dialog

Offset:  
1

Stride:  
1

OK Cancel Help Apply

### Copy from

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

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

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

---

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

---

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

### **Specify address source**

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

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

### **Source code symbol**

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

# Memory Copy

---

## Address

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use `hex2dec` to convert the address to the proper format. The following example converts `0x1000` to decimal form.

```
4096 = hex2dec('1000');
```

For this example, you could enter either `4096` or `hex2dec('1000')` as the address.

## Data type

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

## Data length

Specifies the number of elements to copy from the source location. Each element has the data type specified in **Data type**.

## Use offset when reading

When you are reading the input, use this parameter to specify an offset for the input read. The offset value is in elements with the assigned data type. The **Specify offset source** parameter becomes available when you check this option.

## Specify offset source

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



## Offset

**Offset** tells the block whether to copy the first element of the data at the input address or value, or skip one or more values before starting to copy the input to the destination. **Offset** defines how many values to skip before copying the first value to the destination. Offset equal to one is the default value and **Offset** accepts only positive integers of one or greater.

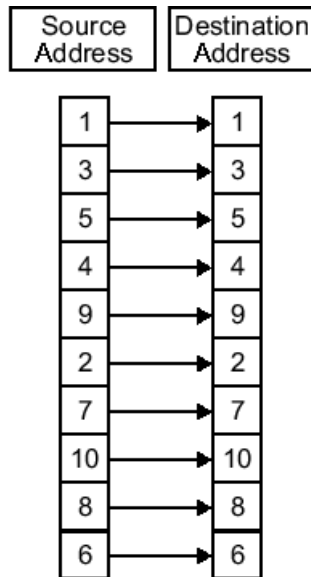
## Stride

Stride lets you specify the spacing for reading the input. By default, the stride value is one, meaning the generated code reads the input data sequentially. When you add a stride value that is not equal to one, the block reads the input data elements not sequentially, but by skipping spaces in the source address equal to the stride. **Stride** must be a positive integer.

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

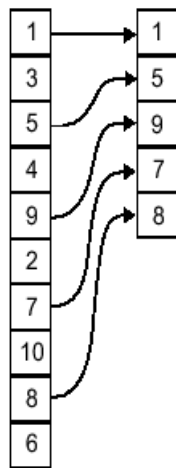
# Memory Copy

---



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

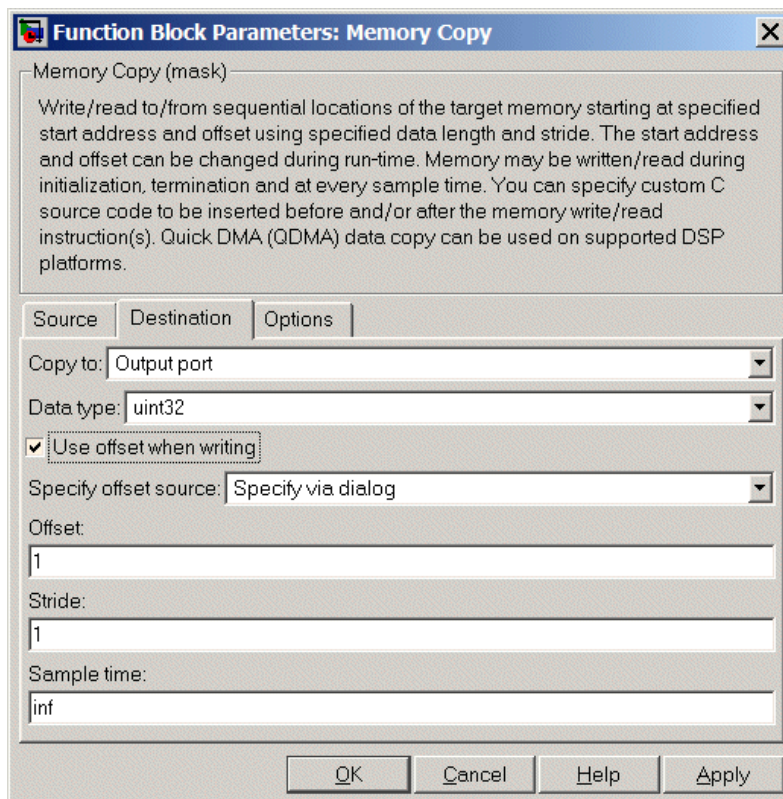
Source Address	Destination Address
----------------	---------------------



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

# Memory Copy

## Destination Parameters



### Copy to

Select the destination for the data. Choose one of the entries on the list:

- **Output port** — Copies the data to the block output port. From the output port the block passes data to downstream blocks in the code.
- **Specified address** — Copies the data to the specified location in **Specify address source** and **Address**.

- **Specified source code symbol** — Tells the block to copy the variable or symbol (variable) to the symbol you enter in **Source code symbol**. When you select this copy to option, you enable the **Source code symbol** parameter.

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

### **Specify address source**

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

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

### **Source code symbol**

Specify the symbol (variable) in the source code symbol table to copy. The symbol table for your program must include this symbol. The block does not verify that the symbol exists and uses valid syntax.

### **Address**

When you select **Specify via dialog** for the address source, you enter the variable address here. Addresses should be in decimal form. Enter either the decimal address or the address as a hexadecimal string with single quotations marks and use **hex2dec** to convert the address to the proper format. This example converts **0x2000** to decimal form.

```
8192 = hex2dec('2000');
```

# Memory Copy

---

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

## Data type

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

## Specify offset source

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

## Offset

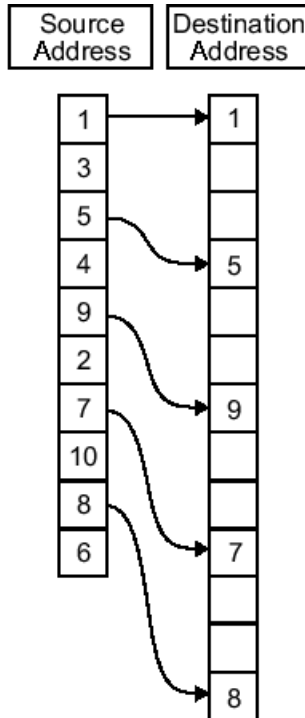
**Offset** tells the block whether to write the first element of the data to be copied to the first destination address location, or skip one or more locations at the destination before writing the output. **Offset** defines how many values to skip in the destination before writing the first value to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

## Stride

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

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

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



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

### Sample time

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

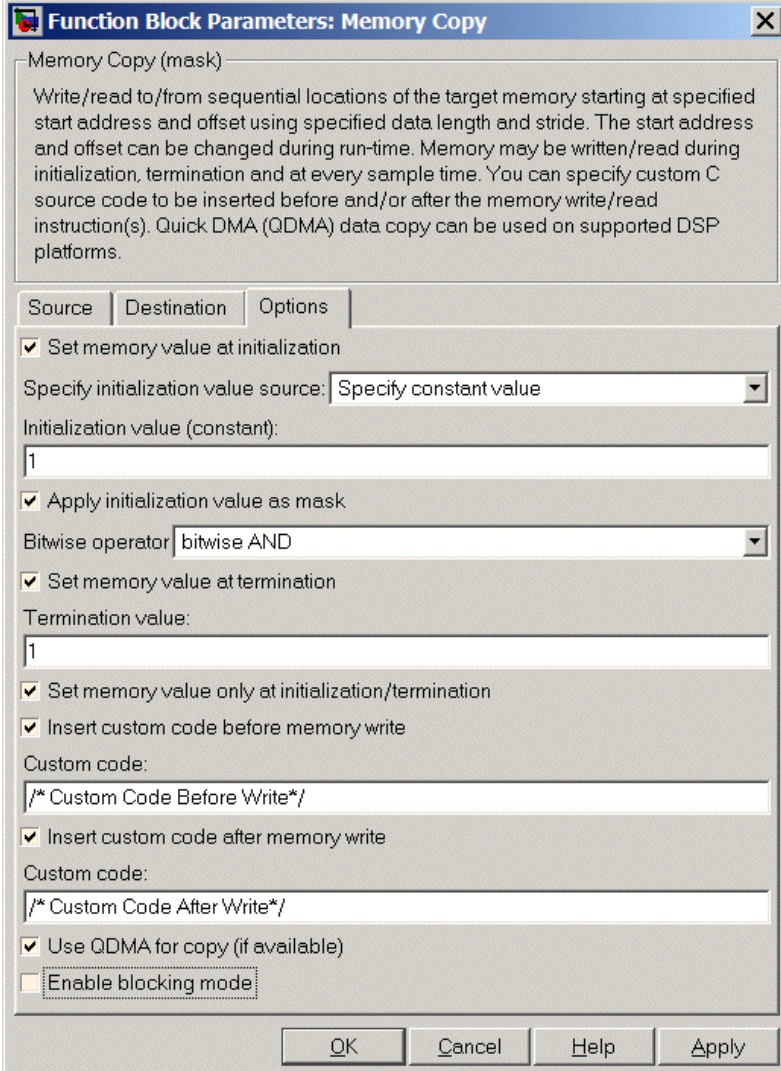
# Memory Copy

---

or the Simulink software model (when there are no input ports on the block). Enter the sample time in seconds as you need.



## Options Parameters



**Function Block Parameters: Memory Copy**

Memory Copy (mask)

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

Source | Destination | Options

Set memory value at initialization

Specify initialization value source: Specify constant value

Initialization value (constant):

1

Apply initialization value as mask

Bitwise operator: bitwise AND

Set memory value at termination

Termination value:

1

Set memory value only at initialization/termination

Insert custom code before memory write

Custom code:

/\* Custom Code Before Write\*/

Insert custom code after memory write

Custom code:

/\* Custom Code After Write\*/

Use QDMA for copy (if available)

Enable blocking mode

OK Cancel Help Apply

## **Set memory value at initialization**

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

## **Specify initialization value source**

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

- **Specify constant value** — Sets a single value to use when your program initializes memory. Enter any value that meets your needs.
- **Specify source code symbol** — Specifies a variable (a symbol) to use for the initial value. Enter the symbol as a string.

## **Initialization value (constant)**

If you check **Set memory value at initialization** and choose **Specify constant value** for **Specify initialization value source**, enter the constant value to use in this field. Any real value that meets your needs is acceptable.

## **Initialization value (source code symbol)**

If you check **Set memory value at initialization** and choose **Specify source code symbol** for **Specify initialization value source**, enter the symbol to use in this field. Any symbol that meets your needs and is in the symbol table for the program is acceptable. When you enter the symbol, the block does not verify whether the symbol is a valid one. If it is not valid you get an error when you try to compile, link, and run your generated code.

## **Apply initialization value as mask**

You can use the initialization value as a mask to manipulate register contents at the bit level. Your initialization value is treated as a string of bits for the mask.

Checking this parameter enables the **Bitwise operator** parameter for you to define how to apply the mask value.

To use your initialization value as a mask, the output from the copy has to be a specific address. It cannot be an output port, but it can be a symbol.

## Bitwise operator

To use the initialization value as a mask, select one of the entries on the following table from the **Bitwise operator** list to describe how to apply the value as a mask to the memory value.

Bitwise Operator List Entry	Description
bitwise AND	Apply the mask value as a bitwise AND to the value in the register.
bitwise OR	Apply the mask value as a bitwise OR to the value in the register.
bitwise exclusive OR	Apply the mask value as a bitwise exclusive OR to the value in the register.
left shift	Shift the bits in the register left by the number of bits represented by the initialization value. For example, if your initialization value is 3, the block shifts the register value to the left 3 bits. In this case, the value must be a positive integer.
right shift	Shift the bits in the register to the right by the number of bits represented by the initialization value. For example, if your initialization value is 6, the block shifts the register value to the right 6 bits. In this case, the value must be a positive integer.

# Memory Copy

---

Applying a mask to the copy process lets you select individual bits in the result, for example, to read the value of the fifth bit by applying the mask.

## **Set memory value at termination**

Along with initializing memory when the program starts to access this memory location, this parameter directs the program to set memory to a specific value when the program terminates.

## **Set memory value only at initialization/termination**

This block performs operations at three periods during program execution—initialization, real-time operations, and termination. When you check this option, the block only does the memory initialization and termination processes. It does not perform any copies during real-time operations.

## **Insert custom code before memory write**

Select this parameter to add custom ANSI C code before the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

## **Custom code**

Enter the custom ANSI C code to insert into the generated code just before the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

## **Insert custom code after memory write**

Select this parameter to add custom ANSI C code immediately after the program writes to the specified memory location. When you select this option, you enable the **Custom code** parameter where you enter your ANSI C code.

## **Custom code**

Enter the custom ANSI C code to insert into the generated code just after the memory write operation. Code you enter in this field appears in the generated code exactly as you enter it.

## **Use QDMA for copy (if available)**

For processors that support quick direct memory access (QDMA), select this parameter to enable the QDMA operation and to access the blocking mode parameter.

If you select this parameter, your source and destination data types must be the same or the copy operation returns an error. Also, the input and output stride values must be one.

## **Enable blocking mode**

If you select the **Use QDMA for copy** parameter, select this option to make the memory copy operations blocking processes. With blocking enabled, other processing in the program waits while the memory copy operation finishes.

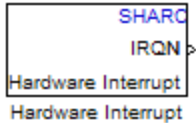
## **See Also**

Memory Allocate

# SHARC Hardware Interrupt

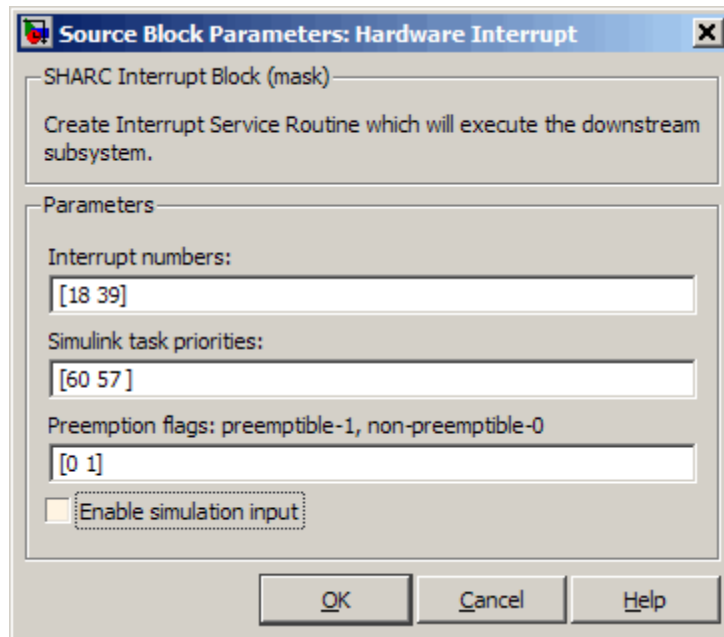
**Purpose** Generate Interrupt Service Routine

**Library** Block Library: idelinklib\_avidvsp



**Description** Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that either run the processes that are downstream from this block or trigger an Idle Task block connected to this block.

## Dialog Box



## Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The valid ranges are 8-36 and 38-40.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. The values in this field and the preemption flag entries in **Preemption flags: preemptible-1, non-preemptible-0** define how the code and processor handle interrupts during asynchronous scheduler operations.

## Simulink task priorities

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Proper code generation requires rate transition code (refer to Rate Transitions and Asynchronous Blocks in the Real-Time Workshop documentation). The task priority values ensure absolute time integrity when the asynchronous task must obtain real time from its base rate or its caller. Typically, assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

## Preemption flags preemptible – 1, non-preemptible – 0

Higher-priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

- Entering 1 indicates that the interrupt can be preempted.
- Entering 0 indicates the interrupt cannot be preempted.

When **Interrupt numbers** contains more than one interrupt value, you can assign different preemption flags to each interrupt by entering a vector of flag values to correspond to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers**

# SHARC Hardware Interrupt

---

contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

In the default settings [0 1], the interrupt with priority 18 in **Interrupt numbers** is not preemptible and the priority 39 interrupt can be preempted.

## **Enable simulation input**

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

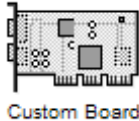


## Purpose

Configure model for a supported processor

## Library

## Description



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

---

**Note** Simulink and Embedded IDE Link software return an error when your model does not include a Target Preferences block or has more than one. When you are generating code for a model, place the Target Preferences block at the top level of your model. When you are generating code for a subsystem, place the Target Preferences block at the subsystem level of your model.

---

The processor options you specify on this block are:

- Processor and board information
- Memory mapping and layout

Setting the options included in this dialog box results in identifying your processor and board to Real-Time Workshop software, Embedded IDE Link, and Simulink software. Setting the options also, configures the memory map for your processor. Both steps are essential for generating code for any board that is custom or explicitly supported.

### Generating Code from Model Subsystems

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

# Target Preferences/Custom Board

---

## Dialog Box

This reference page section contains the following subsections:

- “Board Pane” on page 8-38
- “Memory Pane” on page 8-41
- “Sections Pane” on page 8-44
- “Add Processor Dialog Box” on page 8-46

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

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

## Board Pane

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

### Board type

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

### Processor

Select the type of processor to use from the list. The processor you select determines the contents and setting for options on the **Memory** and **Sections** panes in this dialog box.

## Add New

Clicking **Add new** opens a new dialog box where you specify configuration information for a processor that is not on the Processor list.

For details about the New Processor dialog box, refer to “Add Processor Dialog Box” on page 8-46.

## Delete

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

## CPU Clock (MHz)

Enter the actual clock rate the board uses. The rate you enter in this field does not change the rate on the board. Setting the actual clock rate produces code that runs correctly on the hardware. Setting this value incorrectly causes timing and profiling errors when you run the code on the hardware.

The timer uses the value of **CPU clock** to calculate the time for each interrupt. For example, a model with a sine wave generator block running at 1 kHz uses timer interrupts to generate sine wave samples at the proper rate. For example, using 100 MHz, the timer calculates the sine generator interrupt period as follows:

- Sine block rate = 1 kHz, or 0.001 s/sample
- CPU clock rate = 100 MHz, or 0.000000001 s/sample

To create sine block interrupts at 0.001 s/sample requires:

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

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

# Target Preferences/Custom Board

---

## Board Support

Select the following parameters and edit their values in the text box on the right:

- **Source files** — Enter the full paths to source code files.
- **Include paths** — Add paths to include files.
- **Libraries** — Identify specific libraries for the processor. Required libraries appear on the list by default. To add more libraries, entering the full path to the library with the library file in the text area.
- **Initialize functions** — If your project requires an initialize function, enter it in this field. By default, this parameter is empty.
- **Terminate functions** — Enter a function to run when a program terminates. The default setting is not to include a specific termination function.

---

**Note** Invalid or incorrect entries in these fields can cause errors during code generation. When you enter a file path, library, or function, the block does not verify that the path or function exists or is valid.

---

When entering a path to a file, library, or other custom code, use the following string in the path to refer to the CCS installation directory.

```
$(install_dir)
```

Enter new paths or files (custom code items) one entry per line. Include the full path to the file for libraries and source code.

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

## Operating System

The software disables this option if a supported RTOS is not available for your processor.

## Board name

**Board name** appears after you click **Get from IDE**. Select the board you are using. Match **Board name** with the **Board Type** option near the top of the **Board** pane.

## Processor name

**Processor name** appears after you click **Get from IDE**. If the board you selected in **Board name** has multiple processors, select the processor you are using. Match **Processor name** with the **Processor** option near the top of the **Board** pane.

---

**Note** Click **Apply** to update the board and processor description under **IDE Support**.

---

## Memory Pane

After selecting a board, specify the layout of the physical memory on your processor and board to determine how to use it for your program. For supported boards, the board-specific Target Preferences blocks set the default memory map.

The **Memory** pane contains memory options for:

- **Physical Memory** — Specifies the processor and board memory map
- **Cache Configuration** — Select a cache configuration where available, such as L2 cache, and select one of the corresponding configuration options, such as 32 kb.

For more information about memory segments and memory allocation, consult the reference documentation for the IDE or processor.

The **Physical Memory** table shows the memory segments (or “memory banks”) available on the board and processor. By default, Target

# Target Preferences/Custom Board

---

Preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured Target Preferences blocks shows the memory segments available on the board, but external to the processor. Target Preferences blocks set default starting addresses, lengths, and contents of the default memory segments.

Click **Add** to add physical memory segments to the **Memory banks** table.

After you add the segment, you can configure the starting address, length, and contents for the new segment.

## **Name**

To change the memory segment name, click the name and type the new name. Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

---

**Note** You cannot rename default processor memory segments (name in gray text).

---

## **Address**

**Address** reports the starting address for the memory segment showing in **Name**. Address entries are in hexadecimal format and limited only by the board or processor memory.

## **Length**

From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs).

## **Contents**

Configure the segment to store Code, Data, or Code & Data. Changing processors changes the options for each segment.

You can add and use as many segments of each type as you need, within the limits of the memory on your processor. Every processor must have a segment that holds code, and a segment that holds data.

## **Add**

Click **Add** to add a new memory segment to the processor memory map. When you click **Add**, a new segment name appears, for example NEWMEM1, in **Name** and on the **Memory banks** table. In **Name**, change the temporary name NEWMEM1 by entering the new segment name. Entering the new name, or clicking **Apply**, updates the temporary name on the table to the name you enter.

## **Remove**

This option lets you remove a memory segment from the memory map. Select the segment to remove on the **Memory banks** table and click **Remove** to delete the segment.

## **Cache (Configuration)**

When the **Processor** on the Board pane supports an L2 cache memory structure, the dialog box displays a table of **Cache** parameters. You can use this table to configure the cache as SRAM and partial cache. Both the data memory and the program share this second-level memory.

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

Some processors support code base memory organization. For example, you can configure part of internal memory as code.

Cache level lets you select one of the available cache levels to configure by selecting one of its configurations. For example, you can select L2 cache level and choose one of its configurations, such as 32 kb.

# Target Preferences/Custom Board

---

## Sections Pane

Options on this pane specify where program sections 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 `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler and some can be custom sections.

For more information about program sections and objects, refer to the online help for your IDE.

Within the Sections pane, you configure the allocation of sections for **Compiler** and **Custom** needs.

This table provides brief definitions of the kinds of sections in the **Compiler sections** and **Custom sections** lists in the pane. All sections do not appear on all lists.

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



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

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

## Default Sections

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

## Description

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

## Placement

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

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

## Custom Sections

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

## Sections

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

# Target Preferences/Custom Board

---

## Placement

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

## Add

Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom sections** list. After typing the new name, click **Apply** to add the new section to the list. You can also click **OK** to add the section to the list and close the dialog box.

## Name

Enter the name of the new section here. To add a new section, click **Add**. Then, replace the temporary name with the name to use. Although the temporary name includes a period at the beginning you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

## Contents

Identify whether the contents of the new section are **Code**, **Data**, or **Any**.

## Remove

To remove a section from the **Custom sections** list, select the section and click **Remove**.

## Add Processor Dialog Box

To add a new processor to the drop down list for the **Processors** option, click the **Add new** button on the **Board** pane. The software opens the **Add Processor** dialog box.

## New Name

Provide a name to identify your new processor. You can use any valid C string value in this field. The name you enter in this field appears on the list of processors after you add the new processor.

If you do not provide an entry for each parameter, Embedded IDE Link returns an error message without creating a processor entry.

## **Based On**

When you add a processor, the dialog box uses the settings from the currently selected processor as the basis for the new one. This parameter displays the currently selected processor.

## **Compiler options**

Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class.

## **Linker options**

Identifies the processor family of the new processor to the compiler. Successful compilation requires this switch. The string depends on the processor family or class.

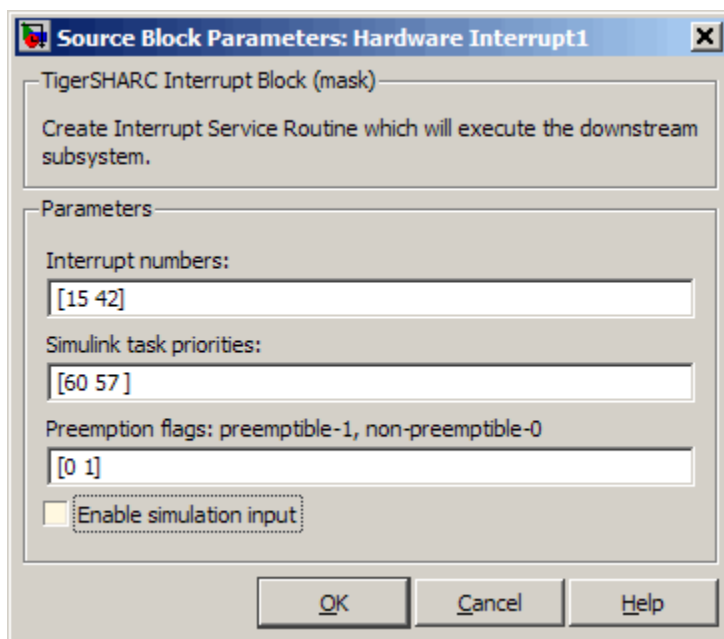
# TigerSHARC Hardware Interrupt

**Purpose** Generate Interrupt Service Routine

**Library** Block Library: idelinklib\_aidvdsp

**Description** Create interrupt service routines (ISR) in the software generated by the build process. When you incorporate this block in your model, code generation results in ISRs on the processor that run the processes that are downstream from the this block or an Idle Task block connected to this block.

## Dialog Box



### Interrupt numbers

Specify an array of interrupt numbers for the interrupts to install. The valid interrupts are 2, 3, 6-9, 14-17, 22-25, 29-32, 37, 38, 41-44, 52.

The width of the block output signal corresponds to the number of interrupt numbers specified in this field. Combined with the **Simulink task priorities** that you enter and the preemption flag you enter for each interrupt, these three values define how the code and processor handle interrupts during asynchronous scheduler operations.

## **Simulink task priorities**

Each output of the Hardware Interrupt block drives a downstream block (for example, a function call subsystem). Simulink model task priority specifies the priority of the downstream blocks. Specify an array of priorities corresponding to the interrupt numbers entered in **Interrupt numbers**.

Simulink model task priority values are required to generate the proper rate transition code (refer to Rate Transitions and Asynchronous Blocks in the Real-Time Workshop documentation). The task priority values are also required to ensure absolute time integrity when the asynchronous task needs to obtain real time from its base rate or its caller. Typically, you assign priorities for these asynchronous tasks that are higher than the priorities assigned to periodic tasks.

## **Preemption flags preemptible – 1, non-preemptible – 0**

Higher priority interrupts can preempt interrupts that have lower priority. To allow you to control preemption, use the preemption flags to specify whether an interrupt can be preempted.

Entering 1 indicates that the interrupt can be preempted. Entering 0 indicates the interrupt cannot be preempted. When **Interrupt numbers** contains more than one interrupt priority, you can assign different preemption flags to each interrupt by entering a vector of flag values, corresponding to the order of the interrupts in **Interrupt numbers**. If **Interrupt numbers** contains more than one interrupt, and you enter only one flag value in this field, that status applies to all interrupts.

# TigerSHARC Hardware Interrupt

---

In the default settings [0 1], the interrupt with priority 15 in **Interrupt numbers** is not preemptible and the priority 42 interrupt can be preempted.

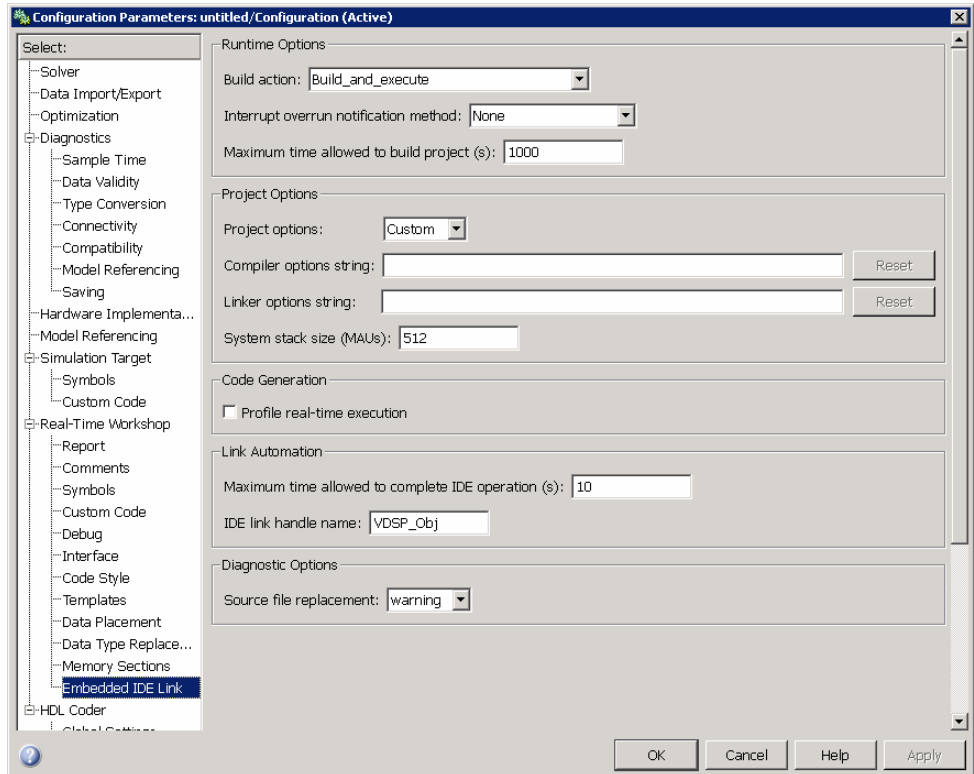
## **Enable simulation input**

When you select this option, Simulink software adds an input port to the Hardware Interrupt block. This port is used in simulation only. Connect one or more simulated interrupt sources to the simulation input.

# Configuration Parameters

---

## Embedded IDE Link Pane



### In this section...

“Overview” on page 9-4

“IDE link handle name” on page 9-6

“Profile real-time execution” on page 9-7

“Profile by” on page 9-9

“Number of profiling samples to collect” on page 9-10

“Project options” on page 9-12

“Compiler options string” on page 9-14



**In this section...**

“Linker options string” on page 9-16

“System stack size (MAUs)” on page 9-18

“Build action” on page 9-19

“Interrupt overrun notification method” on page 9-22

“Interrupt overrun notification function” on page 9-24

“PIL block action” on page 9-25

“Maximum time allowed to build project (s)” on page 9-27

“Maximum time to complete IDE operations (s)” on page 9-29

“Source file replacement” on page 9-31

### **Overview**

Options on this pane configure the generated projects and code for Analog Devices processors. They also enable PIL block generation and provide real-time task execution and stack use profiling.



## IDE link handle name

specifies the name of the `adivdsp` object that the build process creates.

### Settings

**Default:** `VDSP_Obj`

- Enter any valid C variable name, without spaces.
- The name you use here appears in the MATLAB workspace browser to identify the `adivdsp` object.
- The handle name is case sensitive.

### Command-Line Information

**Parameter:** `ideObjName`

**Type:** string

**Value:**

**Default:** `VDSP_Obj`

### Recommended Settings

Application	Setting
Debugging	Enter any valid C program variable name, without spaces
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

### See Also

For more information, refer to [Embedded IDE Link Pane Options](#).

## Profile real-time execution

Enables real-time execution profiling in the generated code by adding instrumentation for task functions or atomic subsystems.

### Settings

**Default:** Off

On  
Adds instrumentation to the generated code to support task execution profiling and generate the profiling report.

Off  
Does not instrument the generated code to produce the profile report.

### Dependencies

This parameter adds **Number of profiling samples to collect**.

Selecting this parameter disables **ID link handle name**.

Setting **Build action** to `Archive_library` or `Create_processor_in_the_loop` project removes this parameter.

### Command-Line Information

**Parameter:** ProfileGenCode

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### Recommended Settings

Application	Setting
Debugging	On
Traceability	On

<b>Application</b>	<b>Setting</b>
Efficiency	No impact
Safety precaution	No impact

### **See Also**

For more information, refer to Embedded IDE Link Pane Options.

## Profile by

Defines how to profile the executing program.

### Settings

**Default:** Task

Task

Profiles model execution by the tasks defined in the model and program.

Atomic subsystem

Profiles model execution by the atomic subsystems in the model.

### Dependencies

Selecting **Profile real-time execution** enables this parameter.

### Command-Line Information

**Parameter:** profileBy

**Type:** string

**Value:** Task | Atomic subsystem

**Default:** Task

### Recommended Settings

Application	Setting
Debugging	Task or Atomic subsystem
Traceability	Archive_library
Efficiency	No impact
Safety precaution	No impact

### See Also

For more information, refer to [Embedded IDE Link Pane Options](#).

For more information about PIL, refer to [Using Processor-in-the-Loop](#).

## Number of profiling samples to collect

Specifies the number of profiling samples to collect. Collection stops when the buffer for profiling data is full.

### Settings

**Default:** 100

**Minimum:** 1

**Maximum:** Buffer capacity in samples

### Tips

- Collecting profiling data on a simulator may take a very long time.
- Data collection stops after collecting the specified number of samples. The application and processor continue to run.

### Dependencies

This parameter is enabled by **Profile real-time execution**.

### Command-Line Information

**Parameter:** ProfileNumSamples

**Type:** int

**Value:** Positive integer

**Default:** 100

### Recommended Settings

Application	Setting
Debugging	100
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact



### **See Also**

For more information, refer to Embedded IDE Link Pane Options.

### Project options

Sets the project options for building your project from the model.

#### Settings

**Default:** Custom

##### Custom

Applies the same settings as the **Release** project configuration in VisualDSP++ software, but does not specify any additional compiler/linker flags. Use this setting to add specialized build and optimization settings.

##### Debug

Applies the **Debug** project options defined by Embedded IDE Link software to the generated project and code. This setting adds a `-g` compiler option, which directs the link software to generate debugging information.

##### Release

Applies the **Release** project configuration defined by Embedded IDE Link software to the generated project and code. This setting adds the following compiler flags:

- `-O` directs the link software to enable optimization
- `-Ov100` directs the link software to optimize for speed by inlining as many functions as possible

#### Dependencies

Selecting Custom disables the **Reset** options for **Compiler options string** and **Linker options string**.

#### Command-Line Information

**Parameter:** projectOptions

**Type:** string

**Value:** Custom | Debug | Release

**Default:** Custom

## Recommended Settings

Application	Setting
Debugging	Custom or Debug
Traceability	Custom, Debug, Release
Efficiency	Release
Safety precaution	No impact

## See Also

For more information, refer to Embedded IDE Link Pane Options.

## Compiler options string

Enter a string of compiler options to define your project configuration.

### Settings

**Default:** No default

### Tips

- To reset the compiler options to the default values, click **Reset**.
- Use spaces between options.
- Verify that the options are valid. The software does not validate the option string.
- Setting **Project options** to **Custom** applies no optimizations to the generated project and code.
- Setting **Project options** to **Debug** applies the `-g` optimization to the generated project and code.
- Setting **Project options** to **Release** applies the `-O` and `-Ov100` optimizations to the generated project and code

### Command-Line Information

**Parameter:** compilerOptionsStr

**Type:** string

**Value:** Custom | Debug | Release

**Default:** Custom

### Recommended Settings

Application	Setting
Debugging	Custom
Traceability	Custom
Efficiency	No impact
Safety precaution	No impact

**See Also**

For more information, refer to Embedded IDE Link Pane Options.

## Linker options string

Enables you to specify linker command options that determine how to link your project when you build your project.

### Settings

**Default:** No default

### Tips

- Use spaces between options.
- Verify that the options are valid. The software does not validate the options string.
- Setting **Project options** to **Release** applies the **-ip** and **-e** optimizations to the generated project and code
- To reset the linker command options to the default values, click **Reset**.

### Dependencies

Setting **Build action** to **Archive\_library** removes this parameter.

### Command-Line Information

**Parameter:** linkerOptionsStr

**Type:** string

**Value:**

**Default:**

### Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

**See Also**

For more information, refer to Embedded IDE Link Pane Options.

## System stack size (MAUs)

Allocates memory for the system stack on the processor.

### Settings

**Default:** 512

**Minimum:** 0

**Maximum:** Available memory

- Enter the stack size in minimum addressable unit (MAUs).
- The software does not verify that your size is valid. Be sure that you enter an acceptable value.

### Dependencies

Setting **Build action** to `Archive_library` removes this parameter.

### Command-Line Information

**Parameter:** `systemStackSize`

**Type:** `int`

**Default:** 512

### Recommended Settings

Application	Setting
Debugging	<code>int</code>
Traceability	<code>int</code>
Efficiency	<code>int</code>
Safety precaution	No impact

### See Also

For more information, refer to [Embedded IDE Link Pane Options](#).



## Build action

Defines how Real-Time Workshop software responds when you press Ctrl+B to build your model.

## Settings

**Default:** Build\_and\_execute

### Build\_and\_execute

Builds your model, generates code from the model, and then compiles and links the code. After linking, this setting downloads and runs the executable on the processor.

### Create\_project

Directs Real-Time Workshop software to create a new project in the IDE.

### Archive\_library

Invokes the IDE Archiver to build and compile your project, but It does not run the linker to create an executable project. Instead, the result is a library project.

### Build

Builds a project from your model. Compiles and links the code. Does not download and run the executable on the processor.

### Create\_processor\_in\_the\_loop\_project

Directs the Real-Time Workshop code generation process to create PIL algorithm object code as part of the project build.

## Dependencies

Selecting Archive\_library removes the following parameters:

- **Interrupt overrun notification method**
- **Interrupt overrun notification function**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Reset**

- **Export IDE link handle to base workspace**

Selecting `Create_processor_in_the_loop_project` removes the following parameters:

- **Interrupt overrun notification method**
- **Interrupt overrun notification function**
- **Profile real-time execution**
- **Number of profiling samples to collect**
- **Linker options string**
- **Reset**
- **Export IDE link handle to base workspace** with the option set to export the handle

### Command-Line Information

**Parameter:** `buildAction`

**Type:** `string`

**Value:** `Build` | `Build_and_execute` | `Create_project Archive_library` | `Create_processor_in_the_loop_project`

**Default:** `Build_and_execute`

### Recommended Settings

Application	Setting
Debugging	<code>Build_and_execute</code>
Traceability	<code>Archive_library</code>
Efficiency	No impact
Safety precaution	No impact

### See Also

For more information, refer to [Embedded IDE Link Pane Options](#).

For more information about PIL, refer to Using Processor-in-the-Loop.

## Interrupt overrun notification method

Specifies how your program responds to overrun conditions during execution.

### Settings

**Default:** None

None

Your program does not notify you when it encounters an overrun condition.

Print\_message

Your program prints a message to standard output when it encounters an overrun condition.

Call\_custom\_function

When your program encounters an overrun condition, it executes a function that you specify in **Interrupt overrun notification function**.

### Tips

- The definition of the standard output depends on your configuration.

### Dependencies

Selecting `Call_custom_function` enables the **Interrupt overrun notification function** parameter.

Setting this parameter to `Call_custom_function` enables the **Interrupt overrun notification function** parameter.

### Command-Line Information

**Parameter:** `overrunNotificationMethod`

**Type:** `string`

**Value:** `None | Print_message | Call_custom_function`

**Default:** `None`

## Recommended Settings

Application	Setting
Debugging	Print_message or Call_custom_function
Traceability	Print_message
Efficiency	None
Safety precaution	No impact

## See Also

For more information, refer to Embedded IDE Link Pane Options.

## Interrupt overrun notification function

Specifies the name of a custom function your code runs when it encounters an overrun condition during execution.

### Settings

No Default

### Dependencies

This parameter is enabled by setting **Interrupt overrun notification method** to `Call_custom_function`.

### Command-Line Information

**Parameter:** `overrunNotificationFcn`

**Type:** string

**Value:** no default

**Default:** no default

### Recommended Settings

Application	Setting
Debugging	String
Traceability	String
Efficiency	No impact
Safety precaution	No impact

### See Also

For more information, refer to [Embedded IDE Link Pane Options](#).

## PIL block action

Specifies whether Real-Time Workshop software builds the PIL block and downloads the block to the processor

### Settings

**Default:** Create\_PIL\_block\_and\_download

Create\_PIL\_block\_build\_and\_download

Builds and downloads the PIL application to the processor after creating the PIL block. Adds PIL interface code that exchanges data with Simulink.

Create\_PIL\_block

Creates a PIL block, places the block in a new model, and then stops without building or downloading the block. The resulting project will not compile in the IDE.

None

Configures model to generate a VisualDSP++ software project that contains the PIL algorithm code. Does not build the PIL object code or block. The new project will not compile in the IDE.

### Tips

- When you click **Build** on the PIL dialog box, the build process adds the PIL interface code to the project and compiles the project in the IDE.
- If you select **Create PIL block**, you can build manually from the block right-click context menu
- After you select **Create PIL Block**, *copy* the PIL block into your model to replace the original subsystem. Save the original subsystem in a different model so you can restore it in the future. Click **Build** to build your model with the PIL block in place.
- *Add* the PIL block to your model to use cosimulation to compare PIL results with the original subsystem results. Refer to the demo “Getting Started with Application Development” in the product demos Embedded IDE Link

- When you select `None` or `Create_PIL_block`, the generated project will not compile in the IDE. To use the PIL block in this project, click **Build** followed by **Download** in the PIL block dialog box.

### Dependency

Enable this parameter by setting **Build action** to `Create_processor_in_the_loop_project`.

### Command-Line Information

**Parameter:** `configPILBlockAction`

**Type:** `string`

**Value:** `None` | `Create_PIL_block` |  
`Create_PIL_block_build_and_download`

**Default:** `Create_PIL_block`

### Recommended Settings

Application	Setting
Debugging	<code>Create_PIL_block_build_and_download</code>
Traceability	<code>Create_PIL_block_build_and_download</code>
Efficiency	<code>None</code>
Safety precaution	No impact

### See Also

For more information about PIL, refer to [Using Processor-in-the-Loop](#).



## Maximum time allowed to build project (s)

Specifies how long, in seconds, the software waits for the project build process to return a completion message.

### Settings

**Default:** 1000

**Minimum:** 1

**Maximum:** No limit

### Tips

- The build process continues even if MATLAB does not receive the completion message in the allotted time.
- This time-out value does not depend on the global time-out value in a `adivdsp` object or the **Maximum time to complete IDE operations** time-out value.

### Dependency

This parameter is disabled when you set **Build action** to `Create_project`.

### Command-Line Information

**Parameter:**TBD

**Type:** int

**Value:** Integer greater than 0

**Default:** 100

### Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

<b>Application</b>	<b>Setting</b>
Efficiency	No impact
Safety precaution	No impact

### **See Also**

For more information, refer to Embedded IDE Link Pane Options.

## Maximum time to complete IDE operations (s)

specifies how long the software waits for IDE functions, such as read or write, to return completion messages.

### Settings

**Default:** 10

**Minimum:** 1

**Maximum:** No limit

### Tips

- The IDE operation continues even if MATLAB does not receive the message in the allotted time. Click **Functions — Alphabetical List** to see a list of the functions and methods.
- This time-out value does not depend on the global time-out value in a `adivdsp` object or the **Maximum time allowed to build project (s)** time-out value

### Command-Line Information

**Parameter:**TBD

**Type:** int

**Value:**

**Default:** 10

### Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

### **See Also**

For more information, refer to Embedded IDE Link Pane Options.

## Source file replacement

Selects the diagnostic action to take if Embedded IDE Link software detects conflicts when you replace source code with custom code.

### Settings

**Default:** warn

none

Does not generate warnings or errors when it finds conflicts.

warning

Displays a warning.

error

Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

### Tips

- The build operation continues if you select `warning` and the software detects custom code replacement problems. You see warning messages as the build progresses.
- Use the `error` setting the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your custom code replacement files.
- Use `none` when you are sure the replacement process is correct and do not want to see multiple messages during your build.

### Command-Line Information

**Parameter:** DiagnosticActions

**Type:** string

**Value:** none | warning | error

**Default:** warning

### Recommended Settings

Application	Setting
Debugging	error
Traceability	error
Efficiency	warning
Safety precaution	error

### See Also

For more information, refer to Embedded IDE Link Pane Parameters.

For more information about custom code replacement, refer to Configuring Custom Code in the *Real-Time Workshop User's Guide*.

# Reported Limitations and Tips

---

## Reported Issues

Some long-standing issues affect the Embedded IDE Link software. When you are using `adivdsp` objects and methods to work with VisualDSP++ software and supported hardware or simulators, recall the information in this section.

The latest issues in the list appear at the bottom. PIL means “processor-in-the-loop” and is similar to hardware-in-the-loop operations.

### **Using 64-bit Symbols in a 64-bit Memory Section on SHARC Processors**

VisualDSP++ compiler design prevents Embedded IDE Link from generating code that accesses 64-bit memory locations correctly. To avoid unexpected results, do not allocate 64-bit data or symbols to 64-bit memory locations on SHARC processors.

When 64-bit data is in 64-bit memory, the compiler generates code that accesses the 64-bit locations as two 32-bit values. Thus, the code does not read and write the 64-bit data correctly. It reads or writes every other 32-bit location, returning or writing incorrect values and possibly exceeding the allocated memory.

Refer to pp. 5-33 in the *ADSP-2136x SHARC Processor Programming Reference, revision 1.0* for a description of how the compiler treats 64-bit (long word) data values.



# Supported Processors

---

This appendix provides the details about the processors, simulators, and software that work with Embedded IDE Link.

## Supported Platforms

In this section...
“Product Features Supported by Each Processor or Family” on page B-2
“Supported Processors and Simulators” on page B-2
“Custom Board Support” on page B-3

This appendix lists the processors and simulators that work with the latest released version of Embedded IDE Link for use with Analog Devices VisualDSP++.

### Product Features Supported by Each Processor or Family

The following table indicates which Embedded IDE Link features are available by processor family.

#### Features by Processor Family

Automation Interface Component		Project Generator Component	Verification	
Processor Family	Debug Mode	Code Generation	Processor-In-the-Loop	Real-Time Execution Profiling
BF52x	Yes	Yes	Yes	Yes
BF531-BF534, BF536-BF539	Yes	Yes	Yes	Yes
SHARC 2136x	Yes	Yes	Yes	Yes
TS20x	Yes	Yes	Yes	Yes

### Supported Processors and Simulators

Embedded IDE Link has been tested on the following processors and boards produced by ADI:

- BF52x
  - ADI Simulators (BF52x)
  - ADSP-BF527 EZ-KIT LITE
- BF531-BF534, BF536-BF539
  - ADI Simulators (BF53x)
  - ADDS-BF537-EZLITE
- SHARC 2136x
  - ADI Simulators (ADSP-2136x Simulator)
  - ADSP-21364 EZ-KIT LITE
  - ADSP-21369 EZ-KIT LITE
- TS 20x
  - ADI Simulators (ADSP-TS201 rev. 1.x/2.x Single Processor Simulator)
  - ADSP-TS201S EZ-KIT LITE

## **Custom Board Support**

You can use Embedded IDE Link with your custom board if:

- The board uses one or more of the supported processors in the preceding list.
- Your processor appears in the Processor list of the Target Preferences or Custom Board block.
- You are able to use Analog Devices VisualDSP++ to interact with your board/processor combination.



## A

- access properties 2-20
- adivdsp 2-18
- adivdsp object properties 2-26
  - procnum 2-25
  - sessionname 2-26
- Analog Devices
  - general code generation options 3-36
  - processor options 3-32
  - run-time options 3-38
  - TLC debugging options 3-35
- Analog Devices model reference 3-53
- Analog Devices processor
  - code generation options 3-38
- Archive\_library 3-54
- asynchronous scheduling 3-3

## B

- block limitations using model reference 3-56
- build configuration
  - compiler options, default 3-44
  - custom 3-44

## C

- compiler options string, set compiler options 3-41
- configuration parameters
  - pane 9-4
    - buildAction 9-19
    - Compiler options string: 9-14
    - configPILBlockAction 9-25
    - DiagnosticActions 9-31
    - gui item name 9-10 9-27 9-29
    - IDE link handle name: 9-6
    - Interrupt overrun notification
      - function: 9-24
    - Linker options string: 9-16
    - overrunNotificationMethod 9-22
    - Profile real-time execution 9-7

- profileBy 9-9
  - projectOptions 9-12
  - System stack size (MAUs): 9-18
- configure the software timer 8-39
- CPU clock speed 8-39
- create custom target function library 3-52
- Create\_project option 3-38
- current CPU clock speed 8-39
- custom compiler options 3-44
- custom project options 3-44

## D

- debug operation
  - new 6-27
- default compiler options 3-44
- default project options 3-44
- discrete solver 3-30

## E

- Embedded IDE Link™ build options
  - Create\_project 3-38
- execution in timer-based models 3-8
- execution profiling
  - subsystem 4-13
  - task 4-10

## F

- file and project operation
  - new 6-27
- fixed-step solver 3-30
- functions
  - overloading 2-23

## G

- generate optimized code 3-38
- getting properties 2-22

**I**

Idle Task block 8-6  
intrinsic. *See* target function library  
issues, using PIL 4-7

**L**

link filters properties  
    getting 2-22  
link properties  
    about 2-25  
    setting 2-22  
link properties, details about 2-25  
linker options string, set linker options 3-42  
linking objects  
    quick reference 2-24  
links  
    closing VisualDSP++® 2-16  
    details 2-25  
    loading files into VisualDSP++® IDE 2-9  
    working with your processor 2-10

**M**

Memory Allocate block 8-9  
Memory Copy block 8-15  
model execution 3-3  
model reference 3-53  
    about 3-53  
    Archive\_library 3-54  
    block limitations 3-56  
    modelreferencecompliant flag 3-56  
    setting build action 3-54  
    target preferences blocks 3-55  
    using 3-54  
model schedulers 3-3  
modelreferencecompliant flag 3-56

**O**

object  
    adivdsp 2-18  
object properties  
    about 2-24  
    quick reference table 2-24  
objects  
    creating objects for VisualDSP++® IDE 2-7  
    introducing the objects for VisualDSP++®  
        IDE tutorial 2-2  
    selecting processors for VisualDSP++® IDE  
        2-6  
    tutorial about using Automation Interface  
        for VisualDSP++® IDE 2-2  
optimization, processor specific 3-38  
overloading 2-23

**P**

PIL block 4-4  
PIL cosimulation  
    overview 4-3  
PIL issues 4-7  
processor configuration options  
    overrun action 3-39  
processor function library. *See* target function  
    library  
processor specific optimization 3-38  
procnum 2-25  
profiling execution  
    by subsystem 4-13  
    by task 4-10  
project options  
    compiler options string 3-41  
    default 3-44  
    linker options string 3-42  
    stack size 3-42  
properties  
    object properties 2-24  
    referencing directly 2-22

- retrieving 2-20
  - function for 2-22
- retrieving by direct property referencing 2-22
- setting 2-20

## R

- Real-Time Workshop options
  - generate code only 3-34
- Real-Time Workshop solver options 3-30
- run-time options
  - build action 3-38
  - overrun action 3-39

## S

- sessionname 2-26
- set overrun action, overrun action 3-39
- set properties 2-20
- set stack size 3-42
- solver option settings 3-30
- stack size, set stack size 3-42
- structure-like referencing 2-22
- synchronous scheduling 3-8

## T

- target configuration options
  - system target file 3-33
- target function library

- assessing execution time after selecting a
  - library 3-49
  - create a custom library 3-52
  - optimization 3-46
  - seeing the library changes in your generated code 3-50
  - selecting the library to use 3-48
  - use in the build process 3-47
  - using with link software 3-46
  - viewing library tables 3-52
  - when to use 3-48
- target preferences blocks in referenced models 3-55
- Target Preferences/Custom Board block 8-37
- TFL. *See* target function library
- timeout
  - timeout 2-26
- timer, configure 8-39
- timer-based models, execution 3-8
- timer-based scheduler 3-8
- timing 3-3
- tutorials
  - objects for VisualDSP++® 2-2

## V

- viewing target function libraries 3-52
- VisualDSP++® IDE objects
  - tutorial about using 2-2