

RealView[®] Debugger

Version 1.8

Essentials Guide



RealView Debugger

Essentials Guide

Copyright © 2002-2004 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History		
Date	Issue	Change
April 2002	A	RealView Debugger v1.5 release
September 2002	B	RealView Debugger v1.6 release
February 2003	C	RealView Debugger v1.6.1 release
September 2003	D	RealView Debugger v1.6.1 release for RVDS v2.0
January 2004	E	RealView Debugger v1.7 release for RVDS v2.1
December 2004	F	RealView Debugger v1.8 release for RVDS v2.2

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

RealView Debugger Essentials Guide

Preface

About this book	x
Feedback	xv

Chapter 1 Using the Essentials Guide

1.1 Using this book	1-2
1.2 Using the documentation suite	1-3

Chapter 2 Features of RealView Debugger

2.1 RealView Debugger v1.8	2-2
2.2 Changes between RealView Debugger v1.7 and v1.6.1	2-10
2.3 Changes between RealView Debugger v1.6.1 and v1.6	2-13
2.4 RealView Debugger v1.6	2-14
2.5 Getting more information online	2-17

Chapter 3 About RealView Debugger

3.1 RealView Debugger	3-2
3.2 About the debugging environment	3-4
3.3 Debugging mode	3-7

Chapter 4 RealView Debugger Desktop

4.1 Basic elements of the desktop	4-2
---	-----

4.2	Finding options on the main menu	4-14
4.3	Working with the Code window toolbars	4-17
4.4	Working in the Code window	4-21

Chapter 5

Getting Started with RealView Debugger

5.1	How to use the tutorial	5-2
5.2	Starting the tutorial	5-3
5.3	Starting RealView Debugger	5-4
5.4	Localizing the RealView Debugger interface	5-5
5.5	Connecting to a debug target	5-9
5.6	Working with memory	5-15
5.7	Loading an image ready for debugging	5-19
5.8	Viewing debugger resources	5-22
5.9	Running the image	5-27
5.10	Basic debugging tasks with RealView Debugger	5-28
5.11	Unloading an image	5-48
5.12	Saving the session	5-49
5.13	Disconnecting from a target	5-52
5.14	Exiting RealView Debugger	5-54
5.15	Completing the tutorial	5-57

Chapter 6

Getting Started with RealView Debugger Projects

6.1	Help on creating RealView Debugger projects	6-2
6.2	Setting up project build options for the basic compilation tasks	6-8

Chapter 7

Getting Started with the RealView Debugger Command Line Interface

7.1	About the CLI	7-2
7.2	Entering commands interactively in the GUI	7-5
7.3	Entering batched commands	7-6
7.4	Using the headless debugger	7-10

Appendix A

Configuration Files Reference

A.1	Overview	A-2
A.2	Files in the etc directory	A-3
A.3	Files in the home directory	A-5

Appendix B

Moving from AXD to RealView Debugger

B.1	RealView Debugger configuration	B-2
B.2	RealView Debugger operations	B-6
B.3	Project management in RealView Debugger	B-8
B.4	Comparison of RealView Debugger and AXD commands	B-11
B.5	Converting legacy AXD scripts to RealView Debugger format	B-16

Appendix C	Moving from armsd to RealView Debugger	
C.1	Comparison of RealView Debugger and armsd commands	C-2
C.2	Converting legacy armsd scripts to RealView Debugger format	C-5
Appendix D	Mapping of Menu Options to Toolbar Buttons	
D.1	Main menu to toolbar button mapping	D-2
	Glossary	

Preface

This preface introduces the *RealView® Debugger v1.8 Essentials Guide*, that shows you how to start using RealView Debugger to manage software projects and to debug your application programs. It contains the following sections:

- *About this book* on page x
- *Feedback* on page xv.

About this book

RealView Debugger provides a powerful tool for debugging and managing software projects. This book contains:

- an introduction to the software components that make up RealView Debugger
- a tutorial to create a project and build an executable image
- a step-by-step guide to getting started, making a connection to a target, and loading an image to start a debugging session
- details about ending a debugging session
- a description of the RealView Debugger desktop
- a glossary of terms for users new to RealView Debugger.

Intended audience

This book has been written for developers who are using RealView Debugger to manage development projects targeted at ARM® core-based processors. It assumes that you are an experienced software developer, and that you are familiar with the ARM development tools. It does not assume that you are familiar with RealView Debugger.

This book includes an appendix that contains information for developers moving from *ARM eXtended Debugger* (AXD) to RealView Debugger.

Examples

The examples given in this book have all been tested and shown to work as described. Your hardware and software might not be the same as that used for testing these examples, so it is possible that certain addresses or values might vary slightly from those shown, and some of the examples might not apply to you. In these cases you might have to modify the instructions to suit your own circumstances.

The examples in this book use the programs stored in the \Examples directory in your RealView Developer Suite root installation, that are targeted at ARM core-based processors.

In general, examples use RealView ARMulator® ISS to simulate an ARM core-based debug target. In some cases, examples are given for other debug target systems.

Using this book

This book is organized into the following chapters:

Chapter 1 *Using the Essentials Guide*

Read this chapter for an introduction to using this book. It describes how this book is organized, where to find specific features, and how to use the tutorial and the rest of the documentation suite.

Chapter 2 *Features of RealView Debugger*

Read this chapter for a history of RealView Debugger releases, including a summary of new functionality in RealView Debugger v1.8.

Chapter 3 *About RealView Debugger*

Read this chapter for an introduction to RealView Debugger. This chapter describes the underlying debugger concepts and explains terminology used in the rest of this book.

Chapter 4 *RealView Debugger Desktop*

Read this chapter for a detailed description of the contents of the RealView Debugger desktop.

Chapter 5 *Getting Started with RealView Debugger*

This chapter explains how to begin using RealView Debugger for the first time. This describes how to start RealView Debugger, make a connection, load an image ready to start debugging, and how to perform some basic debugging tasks.

Chapter 6 *Getting Started with RealView Debugger Projects*

This chapter gives an overview of RealView Debugger projects, provides guidance on creating them, and shows how to set up the basic compilation settings for a project.

Chapter 7 *Getting Started with the RealView Debugger Command Line Interface*

This chapter describes how to begin using the RealView Debugger command-line interface (CLI).

Appendixes and Glossary

Appendix A *Configuration Files Reference*

Read this appendix for details of the files created when you install RealView Debugger v1.8. It describes where files are stored and what information each file contains.

Appendix B *Moving from AXD to RealView Debugger*

This appendix is aimed at developers moving from AXD to RealView Debugger. Read this appendix for pointers on how to carry out specific tasks in RealView Debugger to make this transition, for converting scripts, and for a comparison of the AXD commands with equivalent RealView Debugger commands.

Appendix C *Moving from armsd to RealView Debugger*

This appendix is aimed at developers moving from armsd to RealView Debugger. Read this chapter for details on how to convert scripts and for a comparison of the armsd commands with equivalent RealView Debugger commands.

Appendix D *Mapping of Menu Options to Toolbar Buttons*

This appendix shows the mapping of main menu options to the equivalent toolbar buttons.

Glossary

An alphabetically arranged glossary defines the special terms used in this book.

Typographical conventions

The following typographical conventions are used in this book:

<i>italic</i>	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.
<code>monospace</code>	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
<u><code>monospace</code></u>	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
<i><code>monospace italic</code></i>	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
<code>monospace bold</code>	Denotes language keywords when used outside example code.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information.

ARM periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata, addenda, and Frequently Asked Questions.

ARM publications

This book is part of the RealView Debugger documentation suite. Other books in this suite include:

- *RealView Debugger v1.8 User Guide* (ARM DUI 0153)
- *RealView Debugger v1.8 Project Management User Guide* (ARM DUI 0227)
- *RealView Debugger v1.8 Target Configuration Guide* (ARM DUI 0182)
- *RealView Debugger v1.8 Command Line Reference Guide* (ARM DUI 0175)
- *RealView Debugger v1.8 Extensions User Guide* (ARM DUI 0174).

For details on using the *RealView Compilation Tools* (RVCT), see the books in the RVCT documentation suite.

For details on using RealView ARMulator ISS, see the following documentation:

- *RealView ARMulator ISS User Guide* (ARM DUI 0207).

For general information on software interfaces and standards supported by ARM, see *install_directory\Documentation\Specifications\...*

See the datasheet or Technical Reference Manual for information relating to your hardware.

See the following documentation for information relating to the ARM debug interfaces suitable for use with RealView Debugger:

- *RealView® ICE User Guide* (ARM DUI 0155)
- *Multi-ICE® User Guide* (ARM DUI 0048)
- *ARM Agilent Debug Interface User Guide* (ARM DUI 0158).

Other publications

For a comprehensive introduction to ARM architecture see:

Steve Furber, *ARM system-on-chip architecture* (2nd edition, 2000). Addison Wesley, ISBN 0-201-67519-6.

For the definitive guide to the C programming language, on which the RealView Debugger macro and expression language is based, see:

Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language* (2nd edition, 1989). Prentice-Hall, ISBN 0-13-110362-8.

Feedback

ARM Limited welcomes feedback on both RealView Debugger and its documentation.

Feedback on RealView Debugger

If you have any problems with RealView Debugger, submit a Software Problem Report:

1. Select **Help** → **Send a Problem Report...** from the RealView Debugger main menu.
2. Complete all sections of the Software Problem Report.
3. To get a rapid and useful response, give:
 - a small standalone sample of code that reproduces the problem, if applicable
 - a clear explanation of what you expected to happen, and what actually happened
 - the commands you used, including any command-line options
 - sample output illustrating the problem.
4. Email the report to your supplier.

Feedback on this book

If you have any comments on this book, send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1

Using the Essentials Guide

This chapter introduces the *RealView® Debugger Essentials Guide* and explains how to use this book to start working with RealView Debugger. It contains the following sections:

- *Using this book* on page 1-2
- *Using the documentation suite* on page 1-3.

1.1 Using this book

You are recommended to read the chapters in this book to start debugging your images and to learn how to use RealView Debugger quickly. This book describes the minimum tasks required for the new user.

The main part of this book is organized as a tutorial to introduce you to the features of RealView Debugger so that you can start debugging your own images. It describes:

- terminology used throughout the RealView Debugger documentation suite to describe debugging concepts, see Chapter 3 *About RealView Debugger*
- the RealView Debugger desktop, see Chapter 4 *RealView Debugger Desktop*
- how to start RealView Debugger, connect to a debug target, and load and debug an image, see Chapter 5 *Getting Started with RealView Debugger*
- the steps required to create your first project to build an image, see Chapter 1 *Using the Essentials Guide*
- the steps begin using the RealView Debugger command-line interface (CLI), see Chapter 7 *Getting Started with the RealView Debugger Command Line Interface*.

Read Chapter 2 *Features of RealView Debugger* for details of the new features in RealView Debugger v1.8 and information about previous releases.

———— Note ————

If you are upgrading to RealView Debugger from ARM® *eXtended Debugger* (AXD), see Appendix B *Moving from AXD to RealView Debugger* for a checklist of the tasks you might have to complete to get started, for questions about moving to the new debugger, for details on how to convert command scripts, and a comparison of the CLI commands.

If you are upgrading to RealView Debugger from ARM *Symbolic Debugger* (armsd), see Appendix C *Moving from armsd to RealView Debugger* for details on how to convert command scripts and a comparison of the CLI commands.

1.2 Using the documentation suite

The other books that make up the RealView Debugger documentation suite are:

- *RealView Debugger User Guide*
- *RealView Debugger Project Management Guide*
- *RealView Debugger Target Configuration Guide*
- *RealView Debugger Command Line Reference Guide*
- *RealView Debugger Extensions User Guide.*

At the front of each book is a Preface describing how the contents are organized and how information is presented in the chapters. The following description explains how you might use the books:

1. For a comprehensive description of the debugging features available in RealView Debugger, see *RealView Debugger v1.8 User Guide*. This describes, in detail, how to debug your images and how to configure RealView Debugger to customize your working environment. This book also contains examples of debugging software and details shortcuts, and tips, for the developer.

———— **Note** ————

The *RealView Debugger v1.8 User Guide* also contains information for developers using RealView Debugger on non-Windows platforms. See Appendix B *RealView Debugger for Sun Solaris and Red Hat Linux* for details.

2. RealView Debugger uses projects to save your list of files, understand your build model, and maintain a record of your project-level preferences. A project also enables RealView Debugger to save, and load automatically, specified debugging states, for example breakpoints. See *RealView Debugger v1.8 Project Management User Guide* for a full description of project management in RealView Debugger.
3. *RealView Debugger v1.8 Target Configuration Guide* describes how to connect to targets, how to amend existing targets that are set up in the base product, and how to customize your own targets. It also describes how to create the files required to program Flash.
4. If you want to use the RealView Debugger *Command Line Interface* (CLI) to control your debugging tasks, *RealView Debugger v1.8 Command Line Reference Guide* provides a detailed description of every CLI command and includes examples of their use.
5. If you have the appropriate licenses, you can access RealView Debugger extensions, for example multiprocessor debugging mode. These features are described in *RealView Debugger v1.8 Extensions User Guide*.

See the installation notes delivered with your product for details on installing RealView Debugger.

Chapter 2

Features of RealView Debugger

This chapter describes the features of RealView® Debugger v1.8 and highlights changes in functionality between RealView Debugger v1.8 and previous releases, that is RealView Debugger v1.7, v1.6.1, and v1.6. It contains the following sections:

- *RealView Debugger v1.8* on page 2-2
- *Changes between RealView Debugger v1.7 and v1.6.1* on page 2-10
- *Changes between RealView Debugger v1.6.1 and v1.6* on page 2-13
- *RealView Debugger v1.6* on page 2-14
- *Getting more information online* on page 2-17.

2.1 RealView Debugger v1.8

This section describes the changes between RealView Debugger v1.8 and the previous release RealView Debugger v1.7. It contains the following sections:

- *Updated documentation*
- *Trace, Analysis, and Profiling* on page 2-3
- *Support for gcc built images* on page 2-3
- *DSP support* on page 2-4
- *Changes to the GUI* on page 2-4
- *Changes to the CLI* on page 2-8
- *RealView ARMulator ISS support* on page 2-9
- *RealMonitor support* on page 2-9.

2.1.1 Updated documentation

The changes to the documentation includes the following:

- There is a new chapter about programming Flash in the *RealView Debugger v1.8 Target Configuration Guide*. This chapter describes in detail how to create the files required to program Flash with RealView Debugger, whether you are using the Flash devices and boards currently supported by RealView Debugger, or your own custom Flash devices and boards.
- The *RealView Debugger v1.8 Essentials Guide* (this document) now includes:
 - all information that relates to moving from ARM® *eXtended Debugger* (AXD) or ARM *Symbolic Debugger* (armsd) to RealView Debugger
 - background information about RealView Debugger projects, and how to set up the basic compilation tasks for a project
 - details on how to get started using the RealView Debugger CLI
 - an appendix showing the mapping of the main menu options to the toolbar buttons.
- The task-related information from the *RealView Debugger v1.8 Command Line Reference Guide* is now in the chapter that describes getting started using the RealView Debugger CLI in the *RealView Debugger v1.8 Essentials Guide* (this document).
- Enhancements to tracing with RealView Debugger have been documented (see *Trace, Analysis, and Profiling* on page 2-3 for a summary).
- All documents now reflect the enhancements to the RealView Debugger GUI (see *Changes to the GUI* on page 2-4 for a summary).

- The RealView Debugger online help is now in HTML format, and is displayed using your default web browser.

2.1.2 Trace, Analysis, and Profiling

RealView Debugger v1.8 includes enhancements to the Trace and profiling features:

- tracepoints are now categorized as unconditional or conditional
- the Analysis window has changed (see *Changes to the Analysis window* for a summary of the changes)
- the Configure ETM dialog has changed to support the new *Embedded Trace Macrocell™* v3 (ETMv3), ETB11 port widths (24-bit and 32-bit)
- there is a new trace command, TRACEEXTCOND.

Changes to the Analysis window

The changes to the Analysis window include:

- the following new feature
 - display of interleaved inferred register values
 - functionality to sum profiling data over a number of runs
 - rationalization of the window tabs
- the following improvements:
 - block fetching of trace data, to enhance performance
 - changes to column layouts and the addition of new columns
 - new options in the profiling window.

These changes have the following implications:

- sorting of trace data is supported only in the **Profile** tab
- appending trace data is no longer supported.

For full details on tracing with RealView Debugger, see *RealView Debugger v1.8 Extensions User Guide*.

2.1.3 Support for gcc built images

RealView Debugger supports images built with gcc as follows:

- Images built with gcc v3.2 and v3.4 are fully supported.

- Images built with gcc v2.95.3 are supported, but with no stack backtrace. In addition, these images require converting to ELF format using the `coff2elf` utility. You can obtain this utility from the ARM Support web page.

2.1.4 DSP support

For full details of DSP support in RealView Debugger, see *RealView Debugger v1.8 Extensions User Guide*.

2.1.5 Changes to the GUI

The major changes are described in the following sections:

- *Changes to the Code window main menu structure*
- *Toolbar changes* on page 2-6
- *Internationalization is now supported* on page 2-6
- *Changes to pane views* on page 2-7
- *Changes to breakpoints* on page 2-7
- *Changes to tracepoints* on page 2-7
- *Changes to editing facilities* on page 2-8.

Changes to the Code window main menu structure

The Code window main menus have been restructured, with new menus added, and menu options moved to reflect their functionality:

File

The following changes have been made to this menu:

- the image load options are now available on the new **Target** menu
- connections are now available on the new **Target** menu
- logs and journals are now available on the **Tools** menu
- all workspace-related options are now available from the **Workspace** submenu.

————— Note —————

The **Threads** menu option has been removed, because this feature is available using the **Cycle Threads** toolbar button.

Edit	All editing-related functionality is now available from this menu, including copying and pasting, and searching. The changes include the following: - the new Advanced submenu contains the options from the old Format and Editing Controls submenus - the new Go To submenu contains the original Jump options. - removal of some little-used options, such as VI mode.
View	All pane views are now accessible from the main View menu, instead of the New Pane Views submenu. A new Data Navigator option is available to display the Data Navigator pane. The Browse Symbols dialog box is replaced by a Symbol Browser pane, which has enhanced functionality.
Target	This is a new menu that includes all options relating to connections and image loading.
Project	This menu contains all the project-related options available in previous versions of RealView Debugger, but the grouping has changed.
Build	This is a new menu that includes all the build-related options from the Tools menu.
Debug	The following changes have been made to this menu: - the options on the Execution Control are now available on the main Debug menu. - the breakpoint options are combined into a single Breakpoints... submenu - the Debug/Simple Breakpoints submenu options are redistributed between the new Breakpoints submenu and the new Breakpoints → Conditional submenu - the Complex Breakpoints submenus options are now available from the Breakpoints → Hardware... submenu - the Processor Events option is now available from the option Processor Exceptions... - the Include Commands from File... option is now available on the Tools menu.
Tools	The following changes have been made to this menu: - the build-related options are now available on the new Build menu - the workspace options are now available from the File → Workspace menu

- a new **Logs and Journal** submenu is available for opening and closing log and journal files
- the **Include Commands from File...** option has been moved to this menu from the **Debug** menu
- the **New Editor** submenu has been removed.

Help

The following changes have been made to this menu:

- all web-related options are now on the **ARM on the Web** submenu
- the **Full Online Documentation** option has been removed
- access to the RealView Debugger online help is simplified, and is now available through the **RealView Debugger Help** option.

Toolbar changes

The Code window toolbar is now split into separate function-specific toolbars. Each toolbar can be hidden, moved, or floated independently. The toolbars are:

- File
- Edit
- Debug
- Image
- Connect
- Build
- Find.

For more details, see *Working with the Code window toolbars* on page 4-17.

Internationalization is now supported

Internationalization is now supported, so that you can localize the RealView Debugger interface. You can:

- set the language to English (the default), Japanese, or Chinese
- set the default encoding to ASCII (the default), UTF-8, or Locale.

This displays the main menu and various context menu options in the chosen language. For instructions on how to change these settings, see *Localizing the RealView Debugger interface* on page 5-5.

Changes to pane views

The following changes have been made to panes in RealView Debugger (see *Basic elements of the desktop* on page 4-2 for more details):

- You now have greater flexibility over the docking of panes.
- All panes, except for the File Editor pane, can be floated, hidden, and repositioned on the RealView Debugger desktop.
- You can now set the font used in the pane views. You can set the font name, style, size, and script.
- There is a new Data Navigator pane that enables you to quickly locate a module, function, or variable in an image.
- The Browse Symbols dialog box is replaced by a Symbol Browser pane, which has enhanced functionality.
- The **Expand/Collapse Pane** button has been removed from the pane toolbar.

Changes to breakpoints

Breakpoints are no longer categorized as Simple and Complex. They are now categorized as Software and Hardware, and as unconditional or conditional. As a consequence, the names of the various dialog boxes used to set breakpoints have changed. See *Using breakpoints in RealView Debugger* on page 5-30 for more details.

The Set Address/Data Break/Tracepoint dialog box can now only be used to set breakpoints. Therefore, this dialog box is now called Set Address/Data Breakpoint.

Named breakpoints are no longer supported. Although the Named_breaks group is still present in the Project Properties, the **Named...** menu option has been removed. The Named_breaks group is to be removed in a future release.

Changes to tracepoints

The following changes have been made to tracepoints:

- Tracepoints are no longer categorized as Simple and Complex. They are now categorized as unconditional or conditional.
- You can no longer use the Set Address/Data Break/Tracepoint dialog box to set tracepoints. Therefore, this dialog box is now called Set Address/Data Breakpoint.

See the chapter that describes tracing in RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide* for more details.

Changes to editing facilities

The following changes have been made to the editing facilities in RealView Debugger:

- You now have more control over source code coloring. You can:
 - set both the foreground text color and the background color for various code elements
 - choose from a list of color schemes, including Visual Studio and CodeWarrior, that determines the default colors that are used
 - set colors for the additional code elements identifiers, preprocessor keywords, and operators.
- You can now set the font used in the pane views. You can set the font name, style, size, and script.
- The standalone Editor window has been removed. Also, there is no longer support for using personal standalone editors with RealView Debugger.
- You can no longer use Vi mode when editing.

2.1.6 Changes to the CLI

The following changes have been made to the CLI:

- Spaces are no longer allowed in connection names. Therefore, the RealView ICE is now called RealView-ICE.
- A new CLI command PRINTDSM is available to disassemble the contents of target memory to the **Cmd** tab of the Output pane. The disassembly is in the same format as that shown in the **Dsm** tab of the File Editor pane. Therefore, if you have a journal file open, you can output the disassembly to a file.
- A new trace command, TRACEEXTCOND, is available to set a tracepoint that triggers when an instruction in the specified address range is executed.
- The switches /MB and /R have been added to the PRINTVALUE command:
 - /MB enables multibyte character values to be displayed correctly
 - /R prevents the address being displayed when you specify a variable in a loaded image.
- A /S switch has been added to the INCLUDE command to stop the CLI commands being echoed to the display.

- The ETM_CONFIG command now supports the ETB11 port widths.

2.1.7 RealView ARMulator ISS support

Map files are now supported for *RealView ARMulator*® ISS (RVISS) connections through the Simulator Broker interface.

2.1.8 RealMonitor support

You can now use RealMonitor with RealView ICE connections. Also, how you configure a Multi-ICE® connection to use RealMonitor has changed.

For details on how to configure and use RealMonitor with RealView ICE and Multi-ICE, see *RealView Debugger v1.8 Target Configuration Guide*.

2.2 Changes between RealView Debugger v1.7 and v1.6.1

This section describes the changes between RealView Debugger v1.7 and the previous release RealView Debugger v1.6.1. It contains the following sections:

- *Updated documentation*
- *Advanced debugging facilities*
- *RealView ARMulator ISS*
- *Trace, Analysis, and Profiling* on page 2-13
- *Enhanced RTOS support* on page 2-11
- *New GUI elements* on page 2-12.

2.2.1 Updated documentation

The documentation for developers using RealView Debugger on Windows has been updated to include enhancements and new features in RealView Debugger v1.7. See:

- *RealView Debugger Essentials Guide* for updated information for developers moving to RealView Debugger from AXD. See Appendix B *Moving from AXD to RealView Debugger* for details.
- The detailed description of project management in RealView Debugger has been moved from *RealView Debugger User Guide* to a new book called *RealView Debugger Project Management User Guide*.
- *RealView Debugger User Guide* for information for developers using RealView Debugger on non-Windows platforms. See Appendix B *RealView Debugger for Sun Solaris and Red Hat Linux* for details.

2.2.2 Advanced debugging facilities

RealView Debugger v1.7 enables you to connect to a RealView ICE target using RealView ICE.

2.2.3 RealView ARMulator ISS

In RealView Developer Suite (RVDS), *RealView ARMulator ISS* (RVISS) replaces *ARM Developer Suite™* (ADS) ARMulator.

RVISS enables your debugger to connect using the *Remote Debug Interface* (RDI) and RealView Connection Broker interface. With RealView Connection Broker you connect to multiple instance of RVISS, and you can also connect to a remote RVISS that is on a different system to your debugger. For more details, see the *RealView ARMulator ISS User Guide*.

Note

The RDI connection to RVISS in RealView Debugger is deprecated in this release.

2.2.4 Trace, Analysis, and Profiling

RealView Debugger v1.7 includes enhancements to the Trace and profiling features, including changes to the:

- ETM configuration dialog
- way that trace information is displayed so that interleaved source can be viewed
- method for setting tracepoints
- Set Address/Data Break/Tracepoint dialog
- Analysis window (menu changes).

Note

RealView Debugger v1.7 enables you to access Trace and profiling features without having to purchase a separate license. These features are part of the core product.

2.2.5 Enhanced RTOS support

RealView Debugger v1.7 includes enhancements to RTOS awareness and visualization.

Running System Debug

Running System Debug (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. Where supported by your RTOS, RSD enables you to debug threads individually or in groups.

Thread-based breakpoints

RealView Debugger v1.7 enables you to use the Set Address/Data Break/Tracepoint dialog box and the Break/Tracepoints pane to set thread-based breakpoints when running in RSD mode.

RTOS visualization

This release sees new RTOS visualization features. This gives users an improved threads view in the Process Control pane and provides new menus and tabs in the Resource Viewer window.

RTOS CLI commands

RealView Debugger v1.7 includes new RTOS resource commands that enable you to control RTOS awareness, manage breakpoints and resources, and perform operations on RTOS objects.

———— **Note** ————

RealView Debugger v1.7 does not support RTOS resource CLI commands of the form:

`D<resource-list>=expression`

Use `dos_<resource-list>` commands instead.

See the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide* for full details of all the RTOS support provided by RealView Debugger v1.7.

2.2.6 New GUI elements

New toolbar buttons and menu changes mean that RealView Debugger v1.7 users now have quick access to commonly used features. The changes include:

- a new **Thread** menu, available from the main **File** menu, that replicates the drop-down menu from the **Cycle Threads** button
- a new Actions toolbar button to hide or display the Connection Control window
- a new Actions toolbar button to disconnect from a target.
- an addition to the Execution group, on the Actions toolbar, to execute a **Go to Cursor** operation.

RealView Debugger v1.7 also includes:

- the ability to choose which register is used as a stack pointer, indicated by a new Expression Pointer (EP).
- user-specified data display in the Stack pane
- type ahead for navigating sources and images in the Process Control pane
- persistence of source search paths and path mappings through project settings
- a new **Help** button on the Project Control dialog box
- improved error messages.

2.3 Changes between RealView Debugger v1.6.1 and v1.6

This section describes the changes between RealView Debugger v1.6.1 and the previous release, that is RealView Debugger v1.6. It contains the following sections:

- *Updated documentation*
- *Advanced debugging facilities* on page 2-15
- *Trace, Analysis, and Profiling* on page 2-15
- *RealView Debugger downloads* on page 2-16.

2.3.1 Updated documentation

The documentation for developers using RealView Debugger on Windows has been updated to include new features in RealView Debugger v1.6.1. To aid readability, this release is referred to as RealView Debugger v1.6 throughout the rest of the documentation suite.

Note

The *RealView Debugger User Guide* also contains information for developers using RealView Debugger on non-Windows platforms. See Appendix C *RealView Debugger for Sun Solaris and Red Hat Linux* for details.

2.3.2 Advanced debugging facilities

RealView Debugger v1.6.1 enables you to use the *debug communications channel* (DCC) for semihosting where you are connected to a Multi-ICE target.

2.3.3 Trace, Analysis, and Profiling

New in RealView Debugger v1.6.1, call-graph profiling enables you to view analysis information about a chosen function in your program, for example which functions call this function and which functions are called by it. You can view this information as a graphical display.

2.3.4 RealView Debugger downloads

The **Help** menu now contains a new option to give access to the new Updates and Utilities area. See <http://www.arm.com> to access these resources.

2.4 RealView Debugger v1.6

This section describes the features available in RealView Debugger v1.6. It contains the following sections:

- *Multi-core debugging*
- *OS awareness*
- *Extended Target Visibility (ETV)*
- *Advanced debugging facilities* on page 2-15
- *Trace, Analysis, and Profiling* on page 2-15
- *Project manager* on page 2-16
- *RealView Debugger downloads* on page 2-16.

2.4.1 Multi-core debugging

RealView Debugger v1.6 provides a single debug kernel for mixed ARM and DSP debugging. The debugger provides full support for synchronized start and stop, stepping, and cross triggering of breakpoints.

2.4.2 OS awareness

RealView Debugger v1.6 enables you to:

- use RTOS debug including *Halted System Debug* (HSD)
- interrogate and display resources after execution has halted
- access semaphores and queues
- view the status of the current thread or other threads
- customize views of application threads.

2.4.3 Extended Target Visibility (ETV)

RealView Debugger v1.6 provides visibility of targets such as boards and *System-on-Chip* (SoC). Users can configure targets using Board/Chip definition files and preconfigured files are available:

- ARM family files provided as part of the installation
- customer/partner board files provided through www.arm.com.

2.4.4 Advanced debugging facilities

RealView Debugger v1.6 provides standard debug views and advanced debugging features:

- RealView Debugger supports variables of 64-bit type ‘long long’ throughout the user interface.
- There is now support for module statics, that is static variables of non-local scope, in the Call Stack pane.
- RealView Debugger offers a powerful command-line interface and scripting capability that includes macros support, conversion from AXD and armsd, and history lists to record previous actions.
- Users can access a console (headless debugger) driven from the command line or from scripts.
- RealView Debugger includes an editing control called **Tooltip Evaluation** that provides hover-style evaluation in different code views.
- RealView Debugger enables you to position a Memory pane to display a memory region based on the contents of a variable or register in the Register or Watch panes, or in the **Src** tab.
- Users now have greater control over panes in the Code window and the debug views displayed. RealView Debugger provides the option of using a single Code window to display a wide range of data views during debugging (new in v1.6).
- Programming Flash modules are available as standard.
- Memory mapping is enabled if required.
- Colored memory views indicate the type of memory according to memory map settings.

2.4.5 Trace, Analysis, and Profiling

New in RealView Debugger v1.6, Trace, Analysis, and Profiling is enabled by a Trace debug license. Trace support is available for:

- ARM ETM v1.0 (ETM7™ and ETM9™), including On-Chip Trace
- ARM ETM v2.0 (ETM10™) (beta)
- RVISS ETM simulator
- Motorola M56621 On-Chip Trace
- Intel XScale On-Chip Trace.

Trace and Profiling provides full trace support including simple and complex tracepoints and data filtering:

- viewing raw trace
- viewing code trace
- viewing data trace
- viewing disassembly trace
- tracing of function calls
- the profiling of time spent in each function
- the ability to filter captured trace data by field
- the ability to sort captured trace data by field.

You can set tracepoints directly in the source-level view and/or the disassembly-level view. The same functionality is available in the Memory pane so that you can select regions in memory to trace, or trace a specific memory value when it changes.

2.4.6 Project manager

RealView Debugger v1.6 is a fully-featured *Integrated Development Environment* (IDE) including a project manager and build system.

The project manager includes a Configuration Summary window to display the switch string passed to the compiler tools for build target configurations in the current project.

2.4.7 RealView Debugger downloads

ARM provides a range of services to support developers using RealView Debugger.

The **Help** menu gives access to the new Updates and Utilities area. See <http://www.arm.com> to access these resources.

Use this to download RTOS awareness modules, from a range of vendors, to support RTOS developers and provide enhanced support for different hardware platforms through technical information and board description files.

2.5 Getting more information online

The full documentation suite is available online as DynaText and PDF files.

Depending on your installation, select **Programs** → **ARM** from the Windows **Start** menu. From here select:

- **DynaText Documentation** to view the DynaText files.
- **RealView Developer Suite v2.2** → **PDF Documentation** to view the PDF files.

You can also access the DynaText files from the **Help** menu when RealView Debugger is running.

Note

The DynaText and PDF files contain the same information.

The *RealView Debugger v1.8 User Guide* contains information for developers using RealView Debugger on non-Windows platforms. See Appendix B *RealView Debugger for Sun Solaris and Red Hat Linux* for more details.

Chapter 3

About RealView Debugger

This chapter introduces RealView® Debugger. It explains how the debugger provides a development environment for embedded systems applications using the ARM® family of processors.

This chapter contains the following sections:

- *RealView Debugger* on page 3-2
- *About the debugging environment* on page 3-4
- *Debugging mode* on page 3-7.

3.1 RealView Debugger

RealView Debugger enables you to debug your embedded application programs and have complete control over the flow of the program execution so that you can quickly isolate and correct errors.

3.1.1 RealView Debugger concepts and terminology

The following terminology is used throughout the RealView Debugger documentation suite to describe debugging concepts:

Debug target

A piece of hardware or simulator that runs your application program. A hardware debug target might be a single processor, or a development board containing a number of processors.

Connection The link between the debugger program and the debug target.

Single connection access

The RealView Debugger base product enables you to carry out debugging tasks in single-processor debugging mode, that is where there is only one target connection.

Multiprocessor access

RealView Debugger has been developed as a fully-featured debugger for working with multiprocessor debug target systems. Multiprocessor access enables you to maintain one or more connections to debug targets. Multiprocessor access is a separately licensed feature of RealView Debugger.

DSP RealView Debugger has been developed to provide full debugging functions when working with a range of debug target systems including *Digital Signal Processors* (DSPs). DSP-based debugging is a separately licensed feature of RealView Debugger.

RTOS Operating systems provide software support for application programs running on a target. *Real Time Operating Systems* (RTOSs) are operating systems that are designed for systems that interact with real-world activities where time is critical.

Multithreaded operation

RTOS jobs can share the memory of the processor so that each can share all the data and code of the others. These are called *threads*. RealView Debugger enables you to:

- attach Code windows to threads to monitor one or more threads
- select individual threads to display the registers, variables, and code related to that thread
- change the register and variable values for individual threads.

3.2 About the debugging environment

RealView Debugger uses a three-tier environment to debug application programs:

- debugger software
- a debug interface layer, incorporating the *execution vehicles*
- the debug target.

RealView Debugger uses connection information to describe:

- how the debugger connects to the debug target
- information required to use that connection
- what kind of processor the target is using.

Connection information might also include cached copies of processor registers or memory.

This approach means that you can switch between debug targets without having to start a second or third instance of the debugger program.

This section describes the RealView Debugger debugging environment:

- *Components of RealView Debugger*
- *Debug target interface* on page 3-5
- *Persistence information* on page 3-6.

3.2.1 Components of RealView Debugger

RealView Debugger comprises:

Graphical User Interface (GUI)

This gives access to the main features of the debugger, command processing, and the Code windows.

Target Vehicle Server (TVS)

RealView Debugger maintains connections through the *Target Vehicle Server* (TVS) and plugins that support each combination of target processor and execution vehicle. Using plugins, for example a board file (*.brd), and Board/Chip definition files (*.bcd), enables RealView Debugger to enumerate advanced information about your target environment, hardware, or processor.

The TVS contains the basic debugging functionality and forms most of the software making up RealView Debugger, for example, it supports multithreaded operation under an RTOS, and enables tracing and performance profiling. If you have the appropriate license, the TVS provides multiprocessor debugging.

RealView Connection Broker

RealView Connection Broker, `rvbroker.exe`, handles connections to vehicles that reside on the same workstation as RealView Debugger or any other workstation on your network.

RealView Connection Broker operates in two modes:

- Local** Operating as RealView Simulator Broker, this runs on your local workstation and enables you to access local targets.
- Remote** Operating as RealView Network Broker, this runs on a remote workstation and makes specified targets on that workstation available to other workstations connected to the same network.

3.2.2 Debug target interface

RealView Debugger works with either a hardware or a software debug target. An ARM development board communicating through RealView ICE or Multi-ICE® is an example of a hardware debug target system. RealView ARMulator® ISS is an example of a software debug target system.

The debug target interface contains the execution vehicles that process requests from the client tools to the target. A debug interface might be a JTAG interface, a simulator, or a ROM monitor.

RealView ARMulator ISS

RealView ARMulator ISS (RVISS) is an *Instruction Set Simulator* (ISS). It simulates the instruction sets and architecture of ARM processors, together with a memory system and peripherals. You can extend it to simulate other peripherals and custom memory systems.

RVISS runs on the same host computer as the debugger, and includes facilities for communicating with the debugger.

———— Note ————

RVISS is not the same as the *ARM Developer Suite*™ (ADS) ARMulator supplied with previous releases of RealView Debugger.

3.2.3 Persistence information

RealView Debugger maintains persistence information to enable you to halt a debugging session and resume at a later date. This means that the debugger can remember your working environment including:

- current target connections
- loaded images
- open projects
- desktop settings, for example pane selections and window positions.

3.3 Debugging mode

The RealView Debugger base product enables you to debug your images in single connection mode, that is, where there is only one connection.

If you have the appropriate license, you can also debug multiprocessor applications. RealView Debugger supports such multiprocessor debugging by maintaining connections to multiple debug targets through one or more Code windows. When working in multiprocessor debugging mode, you can use one Code window to cycle through the connected targets, or multiple Code windows to view different targets.

Multiprocessor debugging mode is a separately licensed feature of RealView Debugger and is described in detail in *RealView Debugger v1.8 Extensions User Guide*.

Chapter 4

RealView Debugger Desktop

This chapter describes, in detail, the RealView® Debugger desktop. It describes the contents of the default Code window, and explains how to change them. This chapter describes items and options available from the main menu and the toolbars.

It contains the following sections:

- *Basic elements of the desktop* on page 4-2
- *Finding options on the main menu* on page 4-14
- *Working with the Code window toolbars* on page 4-17
- *Working in the Code window* on page 4-21.

4.1 Basic elements of the desktop

This section describes the default Windows desktop that you see when you run RealView Debugger for the first time after installation and highlights any key features that might be different. It contains the following sections:

- *Splash screen*
- *Code window* on page 4-3
- *Default pane configuration* on page 4-5
- *Debug views and panes* on page 4-8
- *Pane controls* on page 4-9
- *Button toolbars* on page 4-11
- *Color Box* on page 4-12
- *Other window elements* on page 4-12.

Note

The *RealView Debugger v1.8 User Guide* also contains desktop-specific information for developers using RealView Debugger on non-Windows platforms. See Appendix B *RealView Debugger for Sun Solaris and Red Hat Linux* for details.

4.1.1 Splash screen

When you start RealView Debugger, the RealView Debugger splash screen is displayed. During this time, the debugger is checking your environment and creating (or updating) configuration files and a working directory.

Use the `-nologo` flag to run RealView Debugger from the command line without a splash screen.

4.1.2 Code window

When you run RealView Debugger for the first time after installation, the RealView Debugger Code window appears as shown in Figure 4-1.

When you exit RealView Debugger, it remembers the configuration of the panes or views by saving the current state in your workspace.

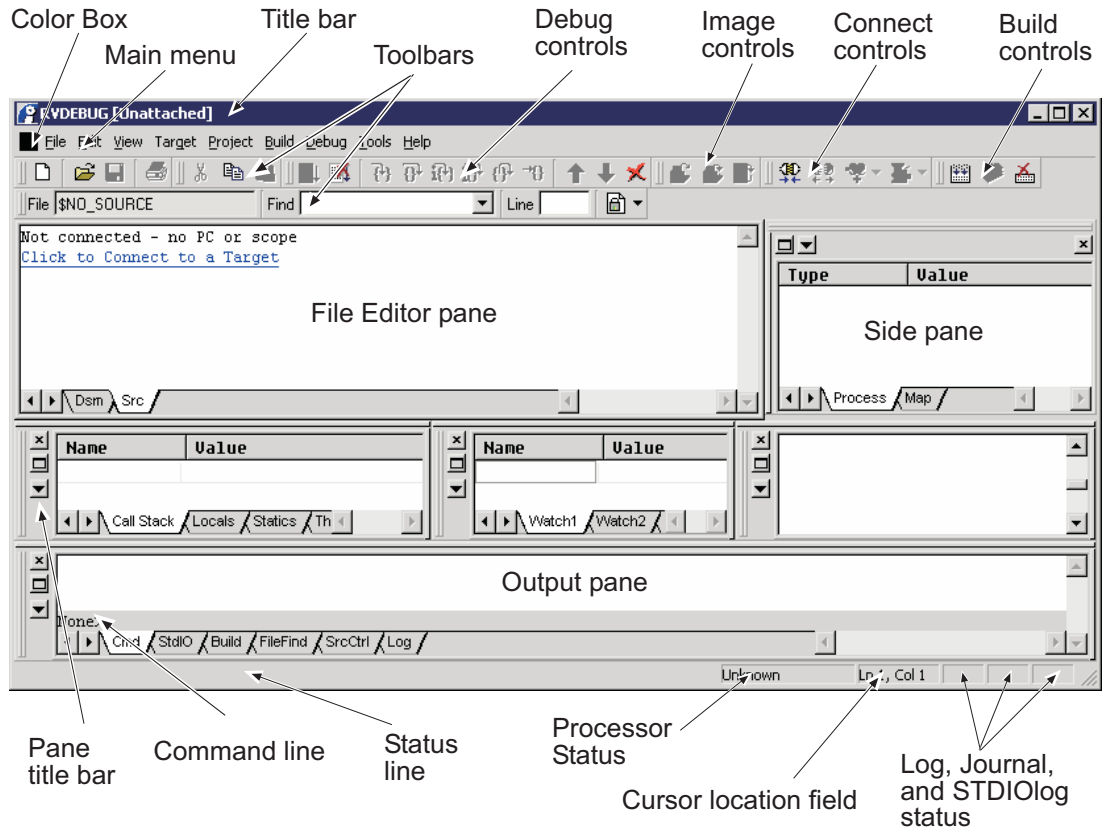


Figure 4-1 Default Code window

The Code window is your main debugging and editing window. The contents of this window change as you:

- connect to targets
- load and unload application programs or files
- configure and customize your working environment.

The appearance of the Code window also depends on your licenses. For example, the base product enables you to debug your images in single connection mode, that is where there is only one connection. If you are licensed to work in multiprocessor debugging mode, the title bar shows your current windows attachment. When you first display the default Code window in multiprocessor mode, this shows [Unattached].

Note

If you are not licensed to use the RealView Debugger multiprocessor extension, some of the buttons in your Code window are disabled. For example, the Connect control buttons for making connections in multiprocessor mode, shown in Figure 4-1 on page 4-3.

Title bar

The Code window title bar gives details of the current project, the current connection, and any processes running on your debug target. In addition to the application icon, the title bar contains (from left to right):

RVDEBUG Identifies the Code window. This changes as you open new windows, for example RVDEBUG_1, or RVDEBUG_2.

(project) The project associated with the image that you loaded.

In RealView Debugger, a project can be associated with a connection, that is it is *bound* to that connection. This is indicated by enclosing the project name in parentheses, for example, (dhrystone).

Where a project is not associated with a particular connection, it is *unbound*. In this case, the project name is enclosed in angled brackets, for example <my_project>.

See the chapter that describes project binding in *RealView Debugger v1.8 Project Management User Guide* for more details.

Also see *Automatic operations performed by a project* on page 6-6.

@endpoint_connection:execution_vehicle

The connection, including the endpoint connection (usually the target processor) and the execution vehicle. For example, if you connect to an ARM7TDMI processor through RealView ICE the title bar shows:

@ARM7TDMI_0:ARM-ARM-NW

See *Elements of the Connection Control window* on page 5-10 for more details on the execution vehicle and endpoint connection.

If you are using an RTOS, and you stop execution, details of the current thread replace the connection details. For example,

T0x19F84_ITCM.ARM

See the chapter that describes RTOS support in the *RealView Debugger v1.8 Extensions User Guide* for details.

[Unattached]

The attachment of the window to a specific connection or thread.

In single-processor debugging mode, this part of the title bar is blank and the option to attach windows to your connection is not available.

If you are working in multiprocessor debugging mode:

- a Code window is unattached by default, shown by [Unattached].
- you can attach a Code window to a specified connection, shown by [Board].

See the chapter that describes working with multiple targets in *RealView Debugger v1.8 Extensions User Guide* for details.

If you are using an RTOS, you can also attach a Code window to a thread. Only the debug information for the attached thread is displayed in that Code window. If a thread is attached to a Code window, the [Unattached] text is removed from the title bar.

See the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide* for details.

4.1.3 Default pane configuration

The default pane configuration in the Code window, shown in Figure 4-1 on page 4-3, contains:

Top pane row

The top pane row contains the File Editor pane and the Process Control pane as a side pane by default.

The File Editor pane is always positioned in the top pane row. However, you can dock one or more other panes to the left or right of the File Editor pane.

File Editor pane

The File Editor pane is always visible when working with RealView Debugger. You cannot replace the File Editor pane with another pane (see *Debug views and panes* on page 4-8). However, you can define how the view is formatted, for example you can change the size of text displayed in the File Editor pane.

Use this area of the Code window to:

- use a shortcut to connect to a target or load an image.
- enter text to create project files
- open source files for editing and resaving
- view disassembly
- set breakpoints to control execution
- use the available menu options to search for specific text as part of debugging
- follow execution through a sequence of source-level and disassembly-level views

The File Editor pane contains a hyperlink to make your first connection to a debug target. When a connection is made, this link changes to give you a quick way to load an image.

When RealView Debugger first starts, the File Editor pane contains tabs to track program execution:

- the **Src** tab shows the current context in the source view
- the **Dsm** tab displays disassembled code with intermixed C/C++ source lines.

If you load an image, or when you are working with source files, more tabs are displayed, for example `dhry_1.c`. In this case, click on the **Src** tab to see the location of the PC.

See the chapter that describes editing source code in *RealView Debugger v1.8 Project Management User Guide* for full details on using the editing facilities in RealView Debugger.

Side pane

By default, this contains the Process Control pane that displays details of the current process when you are connected to target processor. When you first run RealView Debugger, this pane is positioned to the right of the File Editor pane but you can float this pane, and place it at another position on the desktop.

The Process Control pane displays a **Threads** tab if you are debugging an RTOS application.

Middle pane row

The middle pane row contains the following panes by default:

Call Stack pane

Use this pane to:

- display the procedure calling chain from the entry point to the current procedure
- monitor local variables.

The Call Stack pane contains tabs:

Call Stack

Displays the stack functions call chain. This tab is selected by default.

Locals Shows variables local to the current function.

Statics Displays a list of static variables local to the current module.

This Shows objects located by the C++ specific this pointer.

Watch pane

Use this pane to:

- set up variables or expressions to watch
- display current watches
- modify watches already set
- delete existing watches.

The Watch pane contains tabs to display sets of watched values. The first tab, **Watch1**, is selected by default.

Memory pane

Use this pane to:

- display the contents of a range of memory locations on the target
- edit the values stored by the application.

Bottom pane row

The bottom pane row of the default Code window contains the Output pane by default. Select the different tabs to:

- enter commands during a debugging session (**Cmd**)
- handle I/O with your application (**StdIO**)
- see the progress of builds (**Build**)
- see the results of Find in Files operations (**FileFind**)

- see the results of operations using your version control tool (**SrcCtrl**)
- view the results of commands and track events during debugging (**Log**).

The command line is located at the bottom of the Output pane. The command prompt includes the status of the current process, for example, Stop>, Run>, or None> (no process). You can also enter debugger commands at this prompt.

4.1.4 Debug views and panes

RealView Debugger provides a range of debug views accessible through panes:

- Break/Tracepoints
- Call Stack
- Data Navigator
- Memory
- Process Control
- Registers
- Stack
- Symbol Browser
- Watch
- Locals tab of Call Stack pane
- Map tab of Process Control pane
- Threads tab of Process Control pane (with an RTOS extension)
- Output pane.

Substituting panes

All panes except for the File Editor pane can be substituted for another pane. To substitute one pane for another, select the required pane from the **Pane Control** menu of the pane to be substituted (see *Pane controls* on page 4-9 for details).

See the chapter that describes working with debug views in *RealView Debugger v1.8 User Guide* for more details on working with panes.

Adding panes

Select **View** to display the **View** menu to add another pane.

Initially, when you select a pane from the **View** menu it is displayed as a floating pane. If you select a pane that is already visible, then another instance of that pane is displayed, and the name has the suffix *_n*, where *n* is a number starting at one (for example, Memory_1).

If you choose to dock a pane, then a Select Location dialog box is displayed where you can specify the position to place the pane on the desktop.

During your debugging session, you can define which view is displayed in a chosen pane (see *Substituting panes* on page 4-8 for details).

See *Floating, docking, and resizing windows and panes* on page 4-21 for more details on floating and docking panes.

4.1.5 Pane controls

Each configurable pane in the Code window, shown in Figure 4-1 on page 4-3, includes a title bar and pane controls. In the side pane, the pane title bar is displayed horizontally at the top of the pane. In the middle and bottom panes, the title bar is displayed vertically at the left side of the pane.

When you float a pane, the title bar is displayed horizontally at the top of the window, and the title of the window is shown. Docked panes do not show the pane title, but see the description of the gripper bar in this section.

A pane contains the controls:



Pane Control

Click this button to display the **Pane Control** menu where you can change the debug view in the pane and rename the pane.

The top of the menu shows a list of any panes that you have previously viewed and hidden. You can select one of these panes, or select a pane from the **New Pane** submenu.

Visual controls

The visual controls are at the bottom of the **Pane Control** menu. Use these to:

- float the pane if it is docked
- dock the pane if it is floating
- hide the pane (see *Hidden panes* on page 4-10).

See *Floating, docking, and resizing windows and panes* on page 4-21 for more details.



Pane menu Click this button to display the **Pane** menu.

Use this to:

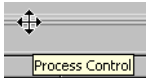
- change the display format
- change how pane contents are updated
- extract data from the pane.

The options available from this menu depend on the pane.



Resize control

Move the mouse to the edge of the pane to resize. When the resize cursor is displayed, click and drag the edge to resize.



Gripper bar

Move the mouse to the edge of the pane containing the gripper bar. In the side pane, the gripper bar is displayed horizontally at the top of the pane. In the middle and bottom panes, the gripper bar is displayed vertically at the left side of the pane.

When the gripper bar cursor is displayed, click and drag the pane to the new position. To float the pane, move the pane outside the RealView Debugger desktop.

If you hover the cursor over the gripper bar, the pane title is displayed as a tooltip. This is not affected by the state of the **Edit** → **Advanced** → **Tooltip Evaluation** option.

Hidden panes

A pane is classed as hidden if you change an existing pane to another type of pane, or you choose to hide it with the **Hide** option on this menu. The hidden state of a pane persists between RealView Debugger sessions.

To remove the hidden status of a pane:

1. Right-click on the toolbar of a pane that is currently displayed.
The context menu shows visible panes as checked, and hidden panes as unchecked.
2. Select the required hidden pane:
 - if the current pane is docked, then the hidden pane is displayed docked, and to the left of the current pane.

- if the current pane is floating, then the hidden pane is displayed floating, and overlays the current pane.
3. Click the **x** button on the pane toolbar to close the pane. That pane is no longer hidden, and so does not appear on any of the pane menus.

Hiding and restoring the Output pane

Only one instance of the Output pane can exist for a Code window. You can hide the Output pane by selecting **Hide** from the **Pane Control** menu. Also, if you click on the **x** button of the Output pane, the Output pane is hidden and not closed.

To make a hidden Output pane visible, either:

- perform steps 1 and 2 of the procedure described in *Hidden panes* on page 4-10
- select **View** → **Output** from the Code window main menu.

Note

The focus is on the tab that was visible at the time Output pane was closed. If the Output pane is open, this option has no effect.

If the Output pane is hidden during program execution, then it is made visible if your program requests user input. The pane is displayed in the same state (floating or docked) as when it was hidden.

4.1.6 Button toolbars

Below the Code window main menu, there are groups of toolbars that provide quick access to many of the features available from menu options, shown in Figure 4-2.



Figure 4-2 Code window toolbar

To hide a toolbar, right-click on any toolbar button. This displays the **Toolbars** menu where you can specify which toolbars are visible.

You can move a toolbar from the default position in the Code window so that it floats on your desktop, for example, the Debug toolbar shown in Figure 4-3. To restore the floating toolbar, double-click anywhere on the toolbar title bar.



Figure 4-3 Debug toolbar (floating)

Repositioning a toolbar in this way applies only to the calling Code window. If you open a new Code window the toolbars are in the default positions.

See *Working with the Code window toolbars* on page 4-17 for details of the buttons available from the Code window toolbar, and how to customize toolbars.

4.1.7 Color Box

Code windows in RealView Debugger are color-coded to help with navigation. This is particularly useful when debugging multiprocessor debug targets and working with multithreaded applications.

The Color Box, shown in Figure 4-4, identifies the connection associated with the current window.

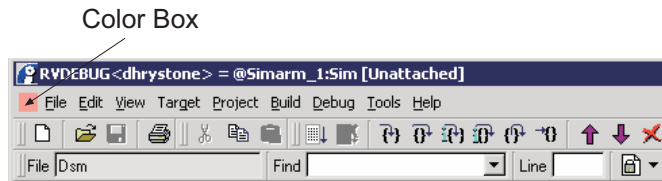


Figure 4-4 Color Box

When you first start RealView Debugger the Code window is not associated with any target or process. The Color Box changes when you make your first connection. As you create new Code windows, these are also color-coded.

Closing your connection changes the Color Box to show that there is no connection associated with the window. Any other Code windows attached to that connection are also updated to match.

When you are working with the Color Box, notice that:

- connection-independent windows, or controls, do not contain a Color Box
- floating panes contain a Color Box that matches the calling window.

4.1.8 Other window elements

There are status display areas at the bottom of the Code window, shown in Figure 4-1 on page 4-3:

Status line As you move through menu options, or when you view button tooltips, this line shows a more detailed explanation of the option or button under the cursor. When loading an image, it also shows the progress of the load.

Processor status

Indicates the processor status, which is one of Unknown, Running, or Stopped.

Cursor location field

As you move the cursor through a file within the File Editor pane, this shows the current location of the text insertion point.

LOG

If this appears to the right of the Cursor location field, this shows that output is being written to a log file.

JOU

If this appears to the right of the Cursor location field, this shows that output is being written to a journal file.

STDIOlog

If this appears to the right of the Cursor location field, this shows that STDIO output is also being written to a STDIOlog file.

4.2 Finding options on the main menu

This section provides a summary of the main menu options available from the Code window (see Figure 4-1 on page 4-3) that enable you to:

- open and close files within the File Editor pane
- manage target images and connections
- manage projects
- build images
- manage workspaces
- navigate, search, and edit source files
- manage new windows and change pane contents
- debug your images
- access the online help system.

Note

Some menu options are not enabled unless you have suitable support, for example the **View** → **Threads tab** option.

The menus available from the main menu bar are:

File	Displays the File menu. The File menu also enables you to examine, and change, your workspace settings. When you are using the File menu the menu option Close Window is not enabled when the default Code window is the only window. This is the main debugging and editing window, and it must be open throughout your debugging session. Close the Code window to close down RealView Debugger.
Edit	Displays the Edit menu. This menu enables you to work with source files as you develop your application. It includes options to: • define how source code is displayed in the File Editor pane. • enable you to work with source files and perform searches on those files as you debug your image • access the various browsers in RealView Debugger.
View	Displays the View menu. This menu enables you to set up new windows and panes as you are working with target connections.

To use custom windows, you must have an appropriate third-party custom window plugin. A custom window plugin can be designed to start automatically when RealView Debugger starts. If you only have plugins of this type, then the **Custom Windows** option remains disabled. Otherwise, if a plugin is designed not to start automatically, then the **Custom Windows** option is enabled. In addition, an option is added to the submenu to enable you to start the custom window plugin. For instructions on installing third-party custom window plugins, and how to use them, see your Vendor documentation.

Target	Displays the Target menu. This menu enables you to connect and disconnect from debug targets, configure your connections, load images, and attach the Code window to a connection.
Project	Displays the Project menu. This menu enables you to work with projects so that you can organize your source files, model your build process, and share files with other developers. This can be used in conjunction with the Build menu to rebuild images. These features are described in full in the chapter that describes working with projects in <i>RealView Debugger v1.8 Project Management User Guide</i>.
Build	Displays the Build menu. This menu enables you to build files, or groups of files, to create your image ready for loading to a target. For details of this menu see the chapter that describes building applications in <i>RealView Debugger v1.8 Project Management User Guide</i>.
Debug	Displays the Debug menu. This menu includes the main facilities you use during a debugging session, such as setting up breakpoints and tracepoints, controlling program execution, and program Flash.
Tools	Displays the Tools menu. This menu enables you to: • set up Analyzer and Tracing features • add and configure macros for use during debugging • log information to the Log, Journal and STDIO log files • run CLI commands from script files • examine, and change, your global configuration options. The Tools menu also provides access to options for Trace, Analysis, and Profiling. These features are described in full in <i>RealView Debugger v1.8 Extensions User Guide</i>.

Note

Supported by selected simulators from Ceva Inc., the **Simulation Control** option is not available in this release.

Help

Displays the **Help** menu.

This menu gives you access to the RealView Debugger online help, to web downloads pages, and displays details of your version of RealView Debugger. You can also use this menu to create and submit a *Software Problem Report* (SPR).

4.3 Working with the Code window toolbars

The Code window toolbars (see Figure 4-2 on page 4-11) give access to many of the features available from the main menu and to additional debugging controls. By default, the Code window shows all toolbars but you can customize which controls are available.

When working with Code window toolbars, use Windows **Tooltips** to see hover-style help when you hold your mouse pointer over a button.

This section includes:

- *File toolbar*
- *Edit toolbar*
- *Debug toolbar*
- *Image toolbar* on page 4-18
- *Connect toolbar* on page 4-18
- *Build toolbar* on page 4-19
- *Find toolbar* on page 4-19
- *Customizing the toolbars* on page 4-20.

4.3.1 File toolbar



Use these buttons when developing applications. They enable you to open files and to save changed files in the File Editor pane. These buttons replicate selected options from the **File** menu.

4.3.2 Edit toolbar



Use these buttons to edit source files in the File Editor pane. They replicate selected options from the **Edit** menu.

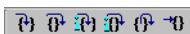
4.3.3 Debug toolbar

These buttons replicate selected options from the **Debug** menu. This toolbar consists of the following controls:



Execution controls

Use these buttons to start and stop program execution.



Stepping controls

Use these buttons to step through a program.



Context controls

Use these buttons to move up and down the stack levels during program execution. These buttons are enabled when an image has been loaded.



Command cancel

Commands submitted to RealView Debugger are queued for execution. Click this button to cancel the last command entered onto the queue.

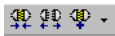
This does not take effect until the previous command has completed.

4.3.4 Image toolbar



Use these buttons to control images. They replicate selected options from the **Target** menu.

4.3.5 Connect toolbar



Use these buttons to control target connections. They replicate selected options from the **Target** menu.

Enabled during a multiprocessor debugging session, click the **Cycle Connections** button to switch to the next available active connection. Click on the drop-down arrow to display a connection menu. This menu lists the active connections with the current connection marked with an asterisk. The menu also provides an option to attach the Code window to the current connection. When a Code window is attached to a connection a tick mark is added to the **Attach Window to Connection** option and the connection that is attached to the Code window. Although you can still cycle through multiple connections, the connection details do not change in Code windows that are attached.

The following button is also part of the Connect toolbar:



Cycle Threads button

Used during a multithreading debugging session, click this button to change to the next active thread. Click on the drop-down arrow to display the list of active threads where you can identify the current thread. The list also shows if the Code window is attached to this thread by adding a tick mark.

This button is only enabled when an underlying operating system is supported. See the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide*.

4.3.6 Build toolbar



Use these buttons to control the build process during your debugging session. They replicate selected options from the **Build** menu.

4.3.7 Find toolbar

This toolbar becomes active when you are editing a file in the File Editor pane. It contains:

File This read-only field shows the name of the file currently displayed in the File Editor pane. If you have changed the file since loading or saving, an asterisk, *, is appended to the end of the filename.

If you are working on several files in the File Editor pane, the File field shows the name displayed on the topmost file tab.

Find This field enables you to perform a quick text search on the file currently displayed in the File Editor pane. Type the required string into the Find field and then press Enter. If you are working on several files in the File Editor pane, the search examines only the file in the topmost file tab. The search behavior is defined by the settings in the Find dialog box displayed from the **Find** menu.

Click on the drop-down arrow to display a list of recently used search strings.

Line Use the Line number field to enter the number of the line where the text insertion point is to be moved.



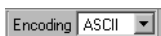
Source control button

This button indicates the read/write status of the current file. Click this button to change the status of a file.

You can edit a file only if the Read-Write icon is displayed (shown here).

Click on the drop-down arrow to access command menu:

- If your sources are not under version control, then this menu enables you to make the current source file in the File Editor pane read-only or editable.
- If your sources are under version control, this button shows the source control commands. See the chapter that describes working with version control systems in the *RealView Debugger v1.8 Project Management User Guide* for more details.



Encoding

Available only when you enable Internationalization. Click the down arrow and select the encoding format you require.

This enables you to view any multibyte strings in a source tab of the File Editor pane in the correct character encoding. It also enables you to search for multibyte text in your sources that matches the selected encoding.

For instructions on how to change the Internationalization settings, see *Localizing the RealView Debugger interface* on page 5-5.

4.3.8 Customizing the toolbars

You can customize the toolbars as described in the following sections:

- *Showing and hiding toolbars*
- *Floating and docking toolbars.*

Showing and hiding toolbars

To show or hide a toolbar:

1. Right-click on any toolbar button or the toolbar background to display the **Toolbars** menu.
Checked options indicate which toolbars are currently shown on the RealView Debugger toolbar. All toolbars are shown by default.
2. Select the toolbar that you want to show or hide. A hidden toolbar is removed from the RealView Debugger toolbar.

———— **Note** ————

If you open a new Code window the toolbars are in the default positions.

Floating and docking toolbars

To float a toolbar:

1. Click on the toolbar control.
2. Drag and drop the toolbar on your desktop.

To dock a floating toolbar, double-click on the toolbar title.

4.4 Working in the Code window

This section describes how to work with the Code window:

- *Floating, docking, and resizing windows and panes*
- *Changing the focus* on page 4-24
- *In-place editing* on page 4-24
- *Working with tabs* on page 4-25
- *Working with scroll bars* on page 4-25.

4.4.1 Floating, docking, and resizing windows and panes

Panes are docked to default positions and have default sizes in the Code window when RealView Debugger starts in the default state.

Resizing docked panes

To change the height of a docked pane in the middle or bottom rows, drag the upper boundary of the row containing the pane to the required height. All panes in the row are resized. This also changes the height of the panes in the first row.

To change the width of a pane, drag the left boundary of the pane to the new position.

Floating a pane

A pane is floating when it is displayed separately from the calling window and can be moved around the desktop. To float a pane:

- select **Float** from the **Pane Control** menu
- click the gripper control, and drag the pane to the required position on your desktop
- double-click on the pane gripper control.

A floating pane is still tied to the calling window, as shown by the Color Box (see *Color Box* on page 4-12).

Docking a pane

To dock a floating pane:

- Select **Dock** from the **Pane Control** menu. If the pane has not been docked previously, a Select Location dialog box is displayed. You can choose to dock the pane to the left, right, or bottom of the Code window.

- click the pane title bar, and drag the pane to the required position on the Code window
- double-click on the pane title bar.

Pane docking behavior

When you dock a pane, the pane is positioned as described in the following sections.

Using the Select Location dialog box to dock a pane

For a pane that has not previously been docked, a Select Location dialog box is displayed containing the following options:

- | | |
|---------------|--|
| Left | The pane is docked to the left of the File Editor pane. If a pane already exists to the left of the File Editor pane, then the pane is docked below that pane. |
| Right | The pane is docked to the right of the File Editor pane. If a pane already exists to the right of the File Editor pane, then the pane is docked below that pane. |
| Bottom | The pane is docked to create a new bottom pane row. |

Dragging and dropping on a Code window border

When you drag and drop a pane on the Code window border:

Left border

The pane is docked as the left-most pane in the top row.

Right border

The pane is docked as the right-most pane in the top row.

Bottom border

The pane is docked to create a new bottom pane row.

Dragging and dropping on a middle or bottom row pane border

When you drag and drop a pane on the border of another pane that is not in the top row:

Left border

The pane is docked in the that row, and to the left of the currently docked pane.

Right border

The pane is docked in the that row, and to the right of the currently docked pane.

Bottom border

The pane is docked in the that row, and above the currently docked pane.

Top border

The pane is docked in the that row, and to the left or right of the currently docked pane, depending on which side of the pane center you drop it.

Dragging and dropping on a top row pane border

When you drag and drop a pane on the border of another pane in the top row:

Left border

The pane is docked in the top row, and to the left of the currently docked pane.

Right border

The pane is docked in the top row, and to the right of the currently docked pane.

Bottom border

The pane is docked in the top row, and below the currently docked pane. However, if you drop it on the bottom of the File Editor pane, then the pane is docked as a new row, between the top row and the row below it.

Top border

The pane is docked in the top row, and above the currently docked pane.

Docking a pane that was previously docked

If the floating pane was previously docked, then the docking behavior depends on whether or not the row that originally contained the pane still exists:

- If the row still exists, then the pane is restored to its original position in that row.
- If the row no longer exists, then the pane is restored to a position in the bottom row. Usually, this is to the left of that row, but this depends on whether or not you have undocked and docked other panes in that row.

4.4.2 Changing the focus

In RealView Debugger, the *focus* indicates the window, or pane, where the next keyboard input takes effect. Use a single left-click on the title bar to move the focus to the Code window, or the pane, where you want to work. If the focus is currently in a source file, you can move the focus from the source file to the Output pane by pressing Ctrl+Tab or Ctrl+Shift+Tab.

When working with several Code windows, left-click inside a pane entry to change the focus. If the pane is floating, the pane title bar changes color to show that it now has the focus, and the title bar of the calling Code window is also highlighted.

If you switch to another Code window by clicking on the title bar, the focus moves to that window and the text insertion point is located inside the File Editor pane. If the context of the Code window is unknown, the text insertion point is located at the command-line prompt.

If you double left-click in a pane entry, for example on the contents of a register in the Register pane, this moves the focus to this pane and highlights the entry ready for editing.

If you right-click in a pane that does not have the focus, the focus does not move to this pane. This action does, however, highlight the chosen entry in the new pane. In this case, use the Code window title bar to see where the focus is currently located.

4.4.3 In-place editing

In-place editing enables you to change a stored value and to see the results of that change instantly. RealView Debugger offers in-place editing whenever possible. For example, if you are displaying the contents of memory or registers and you want to change a stored value:

1. Double-click in the value you want to change, or press Enter if the item is already selected. The value is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Either:
 - enter data to overwrite the highlighted content
 - press the left or right arrow keys to deselect the existing data and position the insertion point where you want to make a change.
3. Press Enter to store the new value in the selected location.

If you press Escape before you press Enter, any changes you have made in the highlighted field are ignored.

In-place editing is not suitable for:

- editing complex data where some prompting is helpful
- editing groups of related items
- selecting values from predefined lists.

In these cases, an appropriate dialog box is displayed.

Note

When using in-place editing, you must either complete the entry and press Enter, or press Escape to cancel the operation. If you move the focus to another pane, RealView Debugger cancels the current in-place editing operation.

4.4.4 Working with tabs

You can access RealView Debugger debugging features using tabbed pages or *tabs*. In the Watch pane, for example, there are multiple tabs and each might show a different set of watched variables. The Output pane contains tabs enabling you to select the view that suits your debugging task.

Right-click on a tab to display text that explains the function of the tab or the content. If you are using the default Windows display settings and you right-click on a tab that is not at the front, the tab name being referenced is colored red for easy identification.

If you right-click over a blank area of the tab bar at the bottom of a window or pane, a context menu enables you to select a tab from a list. You can also display this menu by right-clicking on the left, or right, scroll arrow on the left-hand side of the tab bar.

4.4.5 Working with scroll bars

If you are working in a pane where vertical scroll bars are enabled, right-click on the up, or down, scroll arrow to see the **Scroll** menu. Using this menu you can scroll by a line or a page, or jump to the beginning or end of the related information.

Chapter 5

Getting Started with RealView Debugger

This chapter gives step-by-step instructions to start debugging an application with RealView® Debugger. It is in the form of a tutorial, where each task assumes that you have performed the preceding tasks. This chapter contains the following sections:

- *How to use the tutorial* on page 5-2
- *Starting the tutorial* on page 5-3
- *Starting RealView Debugger* on page 5-4
- *Localizing the RealView Debugger interface* on page 5-5
- *Connecting to a debug target* on page 5-9
- *Working with memory* on page 5-15
- *Loading an image ready for debugging* on page 5-19
- *Viewing debugger resources* on page 5-22
- *Running the image* on page 5-27
- *Basic debugging tasks with RealView Debugger* on page 5-28
- *Unloading an image* on page 5-48
- *Saving the session* on page 5-49
- *Disconnecting from a target* on page 5-52
- *Exiting RealView Debugger* on page 5-54
- *Completing the tutorial* on page 5-57.

5.1 How to use the tutorial

The tutorial describes the main tasks required to begin debugging an application. It assumes that you already have an image to debug. An example Dhrystone project is provided with RealView Developer Suite, which has a prebuilt image. This image is used in the tutorial.

The example Dhrystone project is installed in the directory:

`install_directory\RVD\Examples\...\dhrystone`

The main `...\dhrystone` directory contains:

- The subdirectories `...\Debug` and `...\Release` containing prebuilt debug and release images of `dhrystone.axf`.
- C source files required to build the image.
- Build files, for building the image with RealView Compilation Tools supplied with RealView Developer Suite.
- A RealView Debugger project file, named `dhrystone.prj`, for building the image with RealView Debugger.

The RealView Debugger project file is a user-defined project file. A user-defined project in RealView Debugger is not required for debugging, but it can provide a powerful aid to development.

A project enables RealView Debugger to save your list of files, understand your build model, and maintain a record of your project build preferences.

For a tutorial on how to get started with RealView Debugger projects, see Chapter 6 *Getting Started with RealView Debugger Projects*.

Before you start the tutorial, make a copy of the example Dhrystone project, and use your copy during the tutorial. See *Starting the tutorial* on page 5-3.

5.2 Starting the tutorial

Begin by making a copy of the source files provided so that the tutorial is self-contained and the installed example files are untouched:

1. Create a new directory called `\Tutorial`, in your RealView Developer Suite examples directory:

`install_directory\RVDS\Examples\...\windows\Tutorial`

This is the tutorial project *base directory*.

2. Copy the following files from the examples directory, that is `...\dhrystone`, into your new tutorial directory:
 - C source files `dhry.h`, `dhry_1.c`, and `dhry_2.c`
 - the RealView Debugger project file `dhrystone.prj`
 - the Debug directory.

You can complete this tutorial using the files you have copied. It is not necessary to change any of these files or to amend any configuration files.

5.3 Starting RealView Debugger

To start RealView Debugger, select the following from the Windows **Start** menu:

Programs → ARM → RealView Developer Suite v2.2 → RealView Debugger v1.8

The first time you run RealView Debugger after installation, it creates a unique working directory, in your RealView Debugger home directory, for you to store your personal files, debugger settings, and target configuration files. RealView Debugger then creates files in, or copies files into, this directory ready for your first debugging session.

RealView Debugger uses your login ID as the name of your home directory, which is in the default directory:

```
install_directory\RVD\Core\...\home
```

If a user ID is not available, then RealView Debugger creates a general-purpose directory called *owner*.

———— **Note** ————

You might want to position and resize the main RealView Debugger window to your requirements. Also, resize the panes as required. RealView Debugger remembers the position and size of the window and panes between debugging sessions.

The main RealView Debugger window is known as the Code window (see *Code window* on page 4-3).

5.4 Localizing the RealView Debugger interface

By default, the RealView Debugger interface is configured for the English language. However, you can configure the language for the RealView Debugger interface to one of the following:

- Chinese
- Japanese.

If you do not want to change the internationalization settings, then you can skip this section.

This section includes:

- *Font recommendations*
- *Procedure summary*
- *Configuring the internationalization settings* on page 5-6
- *Configuring the pane views* on page 5-7.

5.4.1 Font recommendations

If you are changing the language to either Chinese or Japanese, then it is recommended that you also change the font as shown in Table 5-1. For instructions on how to change the font, see *Configuring the internationalization settings* on page 5-6.

Table 5-1 Font recommendations

Language	Font to use
Chinese	MingLiU
Japanese	MSGothic or MSMincho

5.4.2 Procedure summary

To localize the RealView Debugger interface, you must:

1. Configure the internationalization settings.
See *Configuring the internationalization settings* on page 5-6 for instructions on how to do this.
2. Configure the panes that display normal text to show the text in the correct text encoding.
See *Configuring the pane views* on page 5-7 for instructions on how to do this.

Images built from C or C++ sources must be compiled with the `Multibyte_chars` project setting enabled (that is, the `--multibyte_chars` compiler option). You can optionally set the `Multibyte_locale` project setting (that is, the `--locale` compiler option). This requires that you either create a RealView Debugger project, or a makefile to build your image:

- for more information about creating RealView Debugger projects, see Chapter 6 *Getting Started with RealView Debugger Projects*
- for more information about using the ARM compiler, see the *RealView Compilation Tools Compiler and Libraries Guide*.

5.4.3 Configuring the internationalization settings

You can set the internationalization settings using:

- Global settings, if you have a shared RealView Debugger environment, and you want all users to use the same configuration. To do this, select the following option from the RealView Debugger main menu:

Tools → Options...

The Options window is displayed.

- Individual workspace settings, for individual users. To do this, select the following option from the RealView Debugger main menu:

File → Workspace → Workspace Options...

The Workspace Options window is displayed.

You can use the following procedure to edit the internationalization settings for both global and workspace settings:

1. Select the `...\rvdebug.ini` group in the left pane of the settings window.
2. Right-click on the **ALL** group in the right pane, and select **Explore** from the context menu.
3. Repeat step 2 for the **Text** group.
4. Select the **Internationalization** group in the left pane.
5. Change the settings to the required values:
 - a. Right-click on the **Enabled** setting, and select **True** from the context menu
 - b. Right-click on the **Language** setting, and select the required language from the context menu, either **English**, **Japanese**, or **Chinese**.

- c. Right-click on the `Default_encoding` setting, and select the required encoding from the context menu, either **ASCII**, **UTF-8**, or **Locale**.
6. Select the `Font_information` group in the left pane.
7. Change the settings to the required values:
 - a. Right-click on the `Pane_font` setting, and select **Edit as font...** from the context menu. The Font dialog box is displayed.
 - b. In the Font dialog box, change the font to that required for your language.
 - c. Set up the remaining font settings as required. For Chinese and Japanese languages, see *Font recommendations* on page 5-5.
 - d. Click **OK** to close the Font dialog box.
8. Select **File** → **Save and Close** to save your changes and close the settings window.
9. Select **File** → **Exit** to exit RealView Debugger.
10. Start RealView Debugger again as described in *Starting RealView Debugger* on page 5-4.

In the Code window, an Encoding toolbar is now displayed, shown in Figure 5-1.

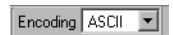


Figure 5-1 Encoding toolbar

5.4.4 Configuring the pane views

To display messages and variable contents correctly in the Code window, you must set the encoding format to the format you specified in the settings window (see *Configuring the internationalization settings* on page 5-6). You can change the format only for the following elements in the Code window:

- File Editor pane, source code tabs, and text searches
- Output pane, all tabs
- individual variables in a Watch pane tab
- individual variables in the **Locals** tab and **Statics** tab of the Call Stack pane.

Changing the encoding format for the File Editor pane and searches

To change the encoding format for the File Editor pane, click the down arrow in the Encoding field on the Find toolbar, and select the required encoding, for example, **UTF-8**.

This also enables you to search your sources for multibyte text that matches the selected format.

Changing the encoding format for the Output pane

To change the encoding format for the Output pane:

1. Click the **Pane** menu for the Output pane.
2. Select **Format...** from the **Pane** menu. A List Selection dialog box is displayed.
3. Select the required encoding from the dialog box, for example, **UTF-8**.
4. Click **OK**.

Changing the encoding format for individual Watch pane entries

To change the encoding for an individual Watch pane entry:

1. Right-click on the Watch entry, and select **Format...** from the context menu. A List Selection dialog box is displayed.
2. Select the required encoding from the dialog box, for example, **UTF-8**.
3. Click **OK**.

Changing the encoding format for individual Call Stack pane entries

To change the encoding for an individual Call Stack pane entry in either the **Locals** tab or the **Statics** tab:

1. Select the required tab in the Call Stack pane, for example **Locals**.
2. Right-click on an entry in the tab, and select **Format...** from the context menu. A List Selection dialog box is displayed.
3. Select the required encoding from the dialog box, for example, **UTF-8**.
4. Click **OK**.

5.5 Connecting to a debug target

Before you can load an image to a debug target, you must connect to the target. You access connections using the Connection Control window. This section contains:

- *How to display the Connection Control window*
- *Elements of the Connection Control window* on page 5-10
- *Making a connection* on page 5-11
- *Configuring your debug target* on page 5-12
- *Setting connect mode* on page 5-13.

For more details on connecting to a debug target, see the chapter that describes connecting to targets in the *RealView Debugger v1.8 Target Configuration Guide*.

For details on working with multiple target connections, see the *RealView Debugger v1.8 Extensions User Guide*.

5.5.1 How to display the Connection Control window

To display the Connection Control window, select **Target** → **Connect to Target...** from the Code window main menu.

Alternatively, click the blue Click to connect to Target hyperlink in the File Editor pane (see Figure 4-1 on page 4-3).

The Connection Control window is displayed (see *Elements of the Connection Control window* on page 5-10 for details). If there are currently no connections to the debug target, the target execution vehicle (for example, ARM-ARM-NW for RealView ICE) is expanded to show the available access-provider connections (for example, RealView-ICE).

————— Note —————



You can also click the **Connection Control** button on the Connect toolbar to display the Connection Control window. If the window is hidden by other windows or minimized, it is displayed as the top-most window on your desktop.

5.5.2 Elements of the Connection Control window

The Connection Control window includes a **Connect** tab. In single processor debugging mode, this is the only tab.

The **Connect** tab includes a tree control, which comprises:



Target execution vehicle

The top-level group is the supported target execution vehicle. It corresponds to the physical connection, and identifies:

- the manufacturer
- the type of physical connection.

For example, for RealView ICE it is ARM-ARM-NW, which indicates that this vehicle is manufactured by ARM® Limited, and is connected across the network.



Access-provider connection

The second-level group shows the type of vehicle, or the debug target interface used to support the connection. For RealView ICE, this is the ARM JTAG debug tool for embedded systems.

Simulator Broker connections can be local or remote:



local Connections to simulators that are installed on the local host machine. By default, these are accessed using the localhost connection.

remote Connections to simulators that are installed on a remote host machine. Although you can modify the localhost connection to access simulators on a remote host, you might want to create a separate connection group, for example remotehost.



When you create another Simulator Broker connection, a different icon is used to distinguish it from the default localhost connection.



Endpoint connections

The third-level entry shows the target processors that are available, for example, ARM7TDMI_0.

The endpoint connection is accompanied by a check box to show the current state of the connection. When connected, this check box is checked. When disconnected, the check box is blank.

Endpoint connections usually have a numerical suffix (for example, _0). If you are licensed to use multiprocessor debugging, each connection is numbered consecutively, starting at zero. This suffix ensures a unique identification for your connections in case you have two processors the

same. For example, if your board contains two ARM7TDMI® processors and an ARM940T processor, then the connections might be named ARM7TDMI_0, ARM7TDMI_1, and ARM940T_2.

Note

If you are licensed to use RealView Debugger extensions, the Connection Control window includes additional tabs. For example, in multiprocessor debugging mode, the window includes a **Connect** tab and a **Synch** tab. See the chapter that describes connecting to targets in *RealView Debugger v1.8 Target Configuration Guide* for full details on how to use this window. Also, see the chapter that describes working with multiple target connections in *RealView Debugger v1.8 Extensions User Guide*.

5.5.3 Making a connection

To connect to a debug target, do the following:

1. If are connecting to a hardware target, make sure your debug hardware and target system are powered on, and configured as described in your debug hardware User Guide.
- 
 2. In the Connection Control window, expand the access-provider connection in the connection tree control to show the endpoint connection (usually the target processor) available for the connection.
- 
 3. Click the check box for the endpoint connection, to connect to the target.

With the connection established, your Code window is updated:

- the Code window title bar is updated with the name of the current connection
- the hyperlink in the File Editor pane changes to enable you to load an image
- the Output pane displays details of the connection software
- panes are updated with debug information, for example the Process Control pane shows process information for the connected target.

If you are always debugging code on the same target you can configure RealView Debugger to make the same connection automatically each time it starts. See the chapter that describes connecting to targets in *RealView Debugger v1.8 Target Configuration Guide* for details of how to set this option.

Example connection to RealView ARMulator ISS

This example shows how to connect to the RealView Connection Broker interface of *RealView ARMulator® ISS* (RVISS):

1. If necessary, expand the Server execution vehicle.

This is the target vehicle that supports RealView Connection Broker connections to local or remote targets. By default, localhost targets are defined.

For details on how to create connections to remote targets, see the chapter that describes working with remote targets in the *RealView Debugger v1.8 Target Configuration Guide*.

2. Expand the localhost access-provider connection.

This includes a new_arm endpoint connection.

3. Click the check box associated with the new_arm connection.

A new endpoint connection is created, called Simarm_1, and the connection information in the Code window title bar changes to:

@Simarm_1:Sim

The default configuration files installed as part of the base product enable you to connect to an ARM7TDMI® core using RVISS on your local workstation.

————— Note —————

If you are licensed to work in multiprocessor debugging mode, another new_arm entry appears for you to make more connections using RVISS.

5.5.4 Configuring your debug target

When you install and configure the software for a debug target, RealView Debugger uses the target device configuration file to determine the number and type of processors. RealView Debugger creates an appropriate endpoint connection for each processor. If you later change the processor configuration, RealView Debugger enables you to access the device configuration file to update the configuration accordingly.

In addition to the device configuration, you can configure additional features for your debug target using the RealView Debugger Connection Properties. To access the Connection Properties, select **Target → Connection Properties...** from the main menu.

Note

You must disconnect from all targets before you configure your target hardware or the associated connection.

Target and connection configuration is not required for you to complete this tutorial. However, you might want to control the way a target processor starts when you connect. To do this, see *Setting connect mode*. Otherwise, continue with *Working with memory* on page 5-15.

If you want to learn more about customizing your targets and connections, see *RealView Debugger v1.8 Target Configuration Guide*. Also, see the device configuration software documentation for your debug target.

5.5.5 Setting connect mode

You can control the way a target processor starts when you connect by setting the connect mode. This is particularly useful when debugging multiprocessor debug targets and working with multiple threads. In single processor debugging mode, you might want to leave an image executing while RealView Debugger closes down and then restart at a later date.

If you are not connected, you can set connect mode when you make a new connection:

1. Select **Target** → **Connect to Target...**, or click the **Connection Control** button, to display the Connection Control window.
2. Right-click on the connection entry, for example `new_arm`, and select **Connect (Defining Mode)...** from the **Connection** context menu.
3. Select the required state from the selection box.

The options listed depend on your target vehicle. For example, when connecting to RVISS, the following options are listed:

- for connections through the RDI interface:
 - **No Reset and Stop** (the default)
 - **No Reset and No Stop** (this is not supported in RVISS)

Note

The RDI connection to RVISS in RealView Debugger is deprecated in this release.

- for connections through the RealView Connection Broker interface:
 - **No Reset and Stop.**

4. Click **OK** to make the connection with the processor in the required state.

———— **Note** ————

If you click **Cancel**, the connection continues using the default connect mode as defined by the target vehicle.

If you set connect mode from the Connection Control window, this temporarily overrides any setting in your target configuration file.

See the chapter that describes connecting to targets in *RealView Debugger v1.8 Target Configuration Guide* for full details of connect mode and how to specify this setting for your debug target.

5.6 Working with memory

Before you load an image, you might have to define memory settings. This depends on the debug target you are using to run your image. For example, if you are using the default RVISS to simulate an ARM processor, setting the value of `top_of_memory` is not appropriate and a default value is used (0x80000).

Where appropriate, defining memory gives you full access to all the memory on your debug target. RealView Debugger enables you to do this in different ways, for example using an include file, or defining the memory map as part of your target configuration settings. For full details of these options, see the chapter that describes memory mapping in *RealView Debugger v1.8 User Guide*.

Note

In the example in this section, you set up memory manually for the current session. Target memory settings defined in this way are only temporary and are lost when you exit RealView Debugger. You can permanently set `top_of_memory` using the Connection Properties window. Configure your debug target so that this setting is used whenever you connect. See the chapter that describes configuring custom targets in *RealView Debugger v1.8 Target Configuration Guide* for an example of how to do this.

This section describes how to set up memory:

- *Using the default setting for top of memory and stack values*
- *Setting top of memory for a session on page 5-16.*

5.6.1 Using the default setting for top of memory and stack values

The `top_of_memory` variable is used to enable the semihosting mechanism to determine the top of stack and heap. If you are not using an ARM architecture-based target, or if your target does not use semihosting, this is ignored.

If you do not set these values, RealView Debugger uses default settings that are dependent on the debug target. For RVISS and ARM processors the default value used for `top_of_memory` is 0x80000.

When you first connect to an ARM architecture-based target, RealView Debugger might display a warning message in the **Cmd** tab, for example:

Warning: No stack/heap or top of memory defined - setting `top_of_memory` to 0x80000.

This means that you have not explicitly set a `top_of_memory` value, and RealView Debugger is using the default value specified in the message. You do not have to explicitly set the `top_of_memory` value to continue with this tutorial. However, if you want to temporarily set the `top_of_memory` value, see *Setting top of memory for a session*. Otherwise, continue with *Loading an image ready for debugging* on page 5-19.

5.6.2 Setting top of memory for a session

If you are working with an appropriate debug target, you can set the `top_of_memory` value on a temporary basis, that is for the current session, using the `@top_of_memory` register.

———— Note ————

If you are using the default RVISS to simulate an ARM processor, this is not a suitable target for setting `top_of_memory` in this way because `top_of_memory` is set from a configuration file rather than from within RealView Debugger.

Setting a top of memory value for a RealView ICE connection

To set the value of `top_of_memory` for an ARM Integrator/AP board and ARM940T core, using RealView ICE:

1. Connect to the RealView ICE debug target as described in *Connecting to a debug target* on page 5-9.
2. Select **Debug** → **Memory/Register Operations** → **Set Register...** to display the Interactive Register Setting dialog box.
3. Specify the register to be changed, `@top_of_memory`, and press Enter to see the current value, shown in Figure 5-2 on page 5-17.
4. Enter the new value, for example `0x40000`, shown in Figure 5-2 on page 5-17.

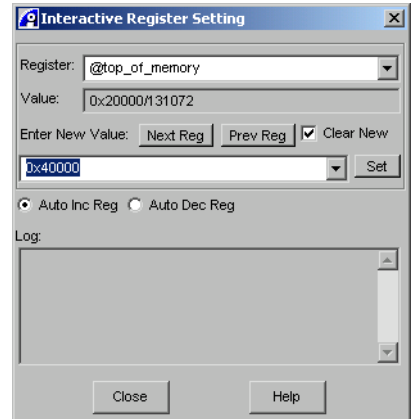


Figure 5-2 Setting top of memory for session

5. Click **Set** to update the register contents. The Log display is updated to record the change.
6. Click **Close** to close the dialog box.
7. Select **View** → **Registers** from the Code window main menu to display the Register pane.

The **Debug** tab in the Register pane, displays the updated value, shown in the example in Figure 5-3.

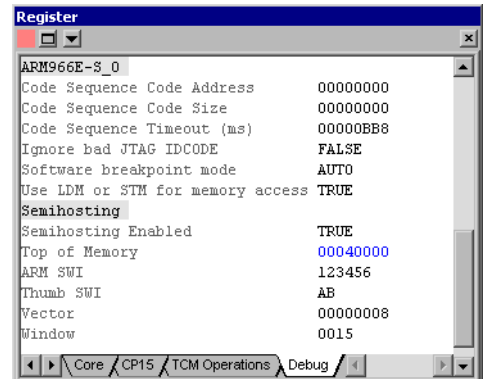


Figure 5-3 Changed settings in the Register pane

The value of `top_of_memory` might be displayed in dark blue to show that it has changed since the last update.

If you set this value too low, loading an image to your target might generate a warning message in the **Cmd** tab:

Warning: No room for heap - could cause unpredictable behavior.

In-place editing

You can use in-place editing to change register contents in the Register pane, for example to change `top_of_memory`:

1. Double-click in the value you want to change. The value is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Type the new value, for example `0x80000`.
3. Press Enter.

If you press Escape before you press Enter, any changes you have made in the highlighted field are ignored.

For full details on setting `top_of_memory` for an ARM architecture-based target, see the chapter that describes memory mapping in *RealView Debugger v1.8 User Guide*.

5.7 Loading an image ready for debugging

When you have connected to a suitably configured debug target you are ready to load your image for debugging. This section describes:

- *Loading an image directly*
- *Loading an image associated with a RealView Debugger project* on page 5-20
- *Loading multi-image applications to a single debug target* on page 5-21
- *Reloading an image* on page 5-21.

5.7.1 Loading an image directly

To load an image directly to your debug target:

1. Select the menu option **Target** → **Load Image...**

Alternatively, click the blue Click to Load Image to Target hyperlink in the File Editor pane.

The Load File to Target dialog is displayed.

———— **Note** ————

Do not change any default settings in the Load File to Target dialog box.

2. Locate the `dhystone.axf` image in your Tutorial directory (see *Starting the tutorial* on page 5-3):

`install_directory\RVDS\Examples\...\Tutorial\Debug`

3. Click **Open**. The image is loaded to the debug target.

In the Code window **Text Coloring** is enabled by default. To turn on line numbering select **Edit** → **Advanced** → **Show Line Numbers** from the Code window main menu.

When you load an image, the debugger:

- inserts the source filename, for the current context, in the File field at the top of the File Editor pane
- highlights the location of the *Program Counter* (PC) at the entry point with a red box
- moves the text insertion point to the current location of the PC
- updates the Code window panes as appropriate, for example, it updates the process information in the Process Control pane

- updates the Code window title bar to show the name of the project associated with the image
- displays the load line in the **Cmd** tab in the Output pane.

If an image is compiled to generate debug information, that is using the `--debug` switch, loading the image enables RealView Debugger to gather debug information about the image and the associated source files. In the Dhrystone example project, this opens the source file `dhry_1.c` into the **Src** tab when you load the image because it has the context. See *Using the code views* on page 5-30 for more details.

5.7.2 Loading an image associated with a RealView Debugger project

To load an image associated with a RealView Debugger project:

1. If you have loaded the Dhrystone image as described in *Loading an image directly* on page 5-19, select the menu option **Project** → **Close Project...** to unload the image.
2. Open the example Dhrystone project:
 - a. Select the menu option **Project** → **Open Project...**. The Select Project to Open dialog is displayed.
 - b. Locate the Dhrystone project file (`dhrystone.prj`) in your Tutorial directory (see *Starting the tutorial* on page 5-3):
`install_directory\RVD\Examples\...\Tutorial`
 - c. Click **Open**. The project is loaded into RealView Debugger.

For an overview of RealView Debugger projects, see Chapter 6 *Getting Started with RealView Debugger Projects*. See *RealView Debugger v1.8 Project Management User Guide* for full details on how to create and manage RealView Debugger projects.

You can also arrange for RealView Debugger to:

- open a project automatically when you connect to a debug target (see *Making a project and a connection interdependent* on page 6-6 for details)
 - perform automatic operations when a project is opened (see *Automatic operations performed by a project* on page 6-6 for details).
3. To load the image to your connected debug target, click the blue Click to Load *image_path* hyperlink in the File Editor pane.

For more details on loading images, see the chapter that describes working with images in the *RealView Debugger v1.8 User Guide*.

5.7.3 Loading multi-image applications to a single debug target

If your application contains multiple images that run on a single target, you can load all the images to the target. Each image must not overlap any of the other images.

To load a multi-image application:

1. Load the first image either:
 - directly (see *Loading an image directly* on page 5-19)
 - from a RealView Debugger project (see *Loading an image associated with a RealView Debugger project* on page 5-20).
2. Load any subsequent images directly.

For each image that you load directly, you must ensure that the **Replace Existing File(s)** check box is not selected on the Load File to Target dialog box.

For more details about working with multiple images, see the chapter that describes working with images in the *RealView Debugger v1.8 User Guide*.

5.7.4 Reloading an image

During your debugging session you might have to amend your source code and then recompile. Select **Target** → **Reload Image to Target** from the Code window to reload an image following these changes.

Reloading an image refreshes any window displays and updates debugger resources.

———— **Note** ————

For full details on loading and reloading images, see the chapter that describes working with images in *RealView Debugger v1.8 User Guide*.

5.8 Viewing debugger resources

RealView Debugger provides a Resource Viewer, which enables you to access all the debugger resources as your debugging session progresses. For example, it enables you to view details about the current connection, and the RTOS resources for any OS-aware connections.

Note

For instructions on how to enable and use RTOS support in RealView Debugger, see the chapter that describes RTOS support in the *RealView Debugger v1.8 Extensions User Guide*. The chapter also describes how to use the Resource Viewer window when working specifically with RTOS resources.

5.8.1 How to display the Resource Viewer window

To display the Resource Viewer window shown in Figure 5-4:

1. Select **View** → **Resource Viewer Window** from the Code window main menu.

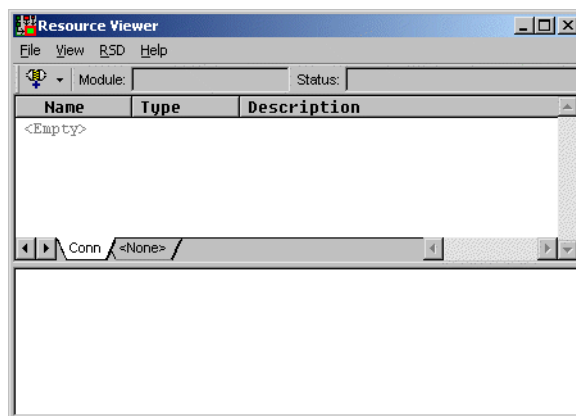


Figure 5-4 Resource Viewer window

Note

Figure 5-4 shows the state of the Resource Viewer window when there are no connections. The contents of your window depend on your debugging mode, licenses, connections, open projects, and windows attachment.

5.8.2 Elements of the Resource Viewer window

When you display the Resource Viewer window, the title bar and the Color Box reflect the calling Code window.

The Resource Viewer window also includes the following elements:

- *File menu*
- *View menu*
- *RSD menu* on page 5-24
- *Help menu* on page 5-25
- *Resources toolbar* on page 5-25
- *Resources list* on page 5-25
- *Details area* on page 5-26.

5.8.3 File menu

This menu contains:

Close Window

Closes the Resource Viewer window.

5.8.4 View menu

This menu contains:

Update List Updates the items displayed in the Resources list.

If you click on the **Conn** tab, choose this option to reread the board file. This might be necessary if you change your connection without closing the Resource Viewer window.

Display Details

Displays details about a selected entry in the Resources list. A short description is shown in the Details area in the window.

You can also display details about an item by double-clicking on the entry in the Resources list.

Display Details as Property

Select this option to display details information in a properties box.

Select this option to change the default display format while the window is open.

Close the Resource Viewer window to restore the default, that is a description is shown in the Details area.

Display List in Log Area

Not available in this release.

Clear Log Clears messages and information displayed in the Details area.

Auto Update Details on Stop

Automatically updates the Details area when any image running on the connection stops.

This gives you information about the state of the connection when the process terminated.

In multiprocessor debugging mode, this applies across all connections.

Auto Update

Automatically updates the Resources list as you change debugger resources. This takes effect when any image running on the connection stops. Selected by default.

Auto Update on Timer...

Not available in this release.

5.8.5 RSD menu

Use this menu to control Running System Debug (RSD):

Enable/Disable RSD

This option enables or disables RSD. The initial state depends on the RSD setting in the RTOS group in the target configuration file.

Stop Target Processor

Stops the target processor and suspends RSD. RealView Debugger switches to HSD mode.

Depending on your configuration settings, click the **Run** button to start execution and restart RSD.

Properties Select this option to see details about the Debug Agent. This includes the status of the RSD module, configuration settings, and RSD breakpoints.

———— Note —————

If RSD is not enabled, some options are grayed out.

5.8.6 Help menu

Select **Help** from the Resource Viewer window menu bar to display the **Help** menu.

This menu gives you access to the RealView Debugger online help and displays details of your version of RealView Debugger. You can also use this menu to create and submit a problem report.

5.8.7 Resources toolbar

This toolbar contains three controls:

Cycle Connections button

Used during a multiprocessor debugging session, click this button to switch to the next available active connection. Changing the active connection updates the information shown in the Resource Viewer window.

Module: This field reports the OS module, for example the name of the RTOS.

Status: This field describes the current RSD status.

5.8.8 Resources list

The Resources list box displays all the resources available to RealView Debugger. In single-processor debugging mode, it contains only the **Conn** tab showing the connection.

Additional resources for RealView Debugger extensions

If you have a multiprocessor debugging license, or you have installed an RTOS extension, additional resources are available.

Multiprocessor resources

In multiprocessor debugging mode, the **Conn** tab shows all active connections. An asterisk (*) indicates the current connection. See the chapter that describes multiprocessor support in *RealView Debugger v1.8 Extensions User Guide* for details.

RTOS resources

With the appropriate RTOS extension, the Resource Viewer window gives access to RTOS resources, for example threads, timers, queues, event flags, and memory in byte and block pools. If you are debugging a multithreaded application, the Resources list box displays these resources in a series of RTOS-specific tabs.

To view your RTOS resources and data structures, you must first configure RTOS support and then execute your image. For full details on using the Resource Viewer window with a multithreaded application, and the RTOS-specific features of this window, see the chapter that describes RTOS in *RealView Debugger v1.8 Extensions User Guide*.

5.8.9 Details area

The bottom half of the Resource Viewer window displays information about a chosen entry, for example a connection or, where applicable, a thread.

5.9 Running the image

To run an image, either:

- Select **Debug** → **Run** from the main menu.
- Click the Run button on the Debug toolbar.



The Code window is updated as follows:

- the processor status changes to Running, and includes an animated progress indicator
- the process information in the Process Control pane is updated to show that the image is running
- the **StdIO** tab of the Output pane is selected, and application messages and prompts are displayed
- the CLI prompt in the **Cmd** tab of the Output pane changes to Run>.

5.10 Basic debugging tasks with RealView Debugger

The basic debugging tasks you can perform with RealView Debugger are shown in Table 5-2. The referenced sections give instructions for these tasks.

See chapter that describes working breakpoints in the *RealView Debugger v1.8 User Guide* for detailed information about using breakpoints.

Because setting a tracepoint is very similar to setting a breakpoint, the use of tracepoints is not described in this section. In addition, the analysis of trace output is too detailed to include in a getting started tutorial. If you want to use the Tracing feature of RealView Debugger, see the chapter that describes tracing with RealView Debugger in the *RealView Debugger v1.8 Extensions User Guide*.

Table 5-2 Basic Debugging Tasks

Task	For instructions, see
Preparing to start debugging	<i>Getting started</i> on page 5-29
Displaying line numbers in the current source view	<i>Displaying line numbers</i> on page 5-29
Examining different code views	<i>Using the code views</i> on page 5-30
Viewing target status	<i>Viewing target status</i> on page 5-30
Viewing hardware breakpoint support	<i>Viewing your hardware breakpoint support</i> on page 5-33
Setting unconditional breakpoints	<i>Setting default breakpoints</i> on page 5-33
Setting conditional breakpoints	<i>Setting conditional breakpoints</i> on page 5-34
Managing breakpoints	<i>Managing breakpoints</i> on page 5-36
Setting watches	<i>Setting watches</i> on page 5-37
Displaying register contents	<i>Displaying register contents</i> on page 5-39
Changing register contents	<i>Changing register contents</i> on page 5-40
Using the Process Control pane	<i>Process control</i> on page 5-40
Displaying memory contents	<i>Displaying memory contents</i> on page 5-42
Displaying variables	<i>Displaying variables</i> on page 5-44
Using Tooltip Evaluation to examine variables and register contents	<i>Tooltip evaluation</i> on page 5-44

Table 5-2 Basic Debugging Tasks (continued)

Task	For instructions, see
Using the call stack	<i>Using the call stack</i> on page 5-44
Using the stack	<i>Using the stack</i> on page 5-45
Using the Data Navigator pane	<i>Using the Data Navigator pane</i> on page 5-46
Using the Symbol Browser pane	<i>Using the Symbol Browser pane</i> on page 5-47

5.10.1 Getting started

Before you can start debugging your image you must complete the following steps:

1. Start RealView Debugger, see *Starting RealView Debugger* on page 5-4.
2. Connect to your target, see *Making a connection* on page 5-11.
3. Set `top_of_memory`, if appropriate for your debug target, see *Setting top of memory for a session* on page 5-16.
4. Load your image, see *Loading an image ready for debugging* on page 5-19.

Continuing with the tutorial

If you started the image running as described in *Running the image* on page 5-27, do the following:

1. Enter `100` at the prompt for the number of runs, then press Enter.
This value is too small for a meaningful run through Dhrystone, but for now it is sufficient to quickly terminate program execution.
2. Reload the image as described in *Reloading an image* on page 5-21.
3. Continue with the debugging tasks described in this section.

5.10.2 Displaying line numbers

To display line numbers for any source file displayed in the File Editor pane, select the menu option:

Edit → Advanced → Show Line Numbers

5.10.3 Using the code views

Use the File Editor pane to view source code during your debugging session. In the example shown in Figure 4-1 on page 4-3, the File Editor pane contains three tabs:

- the **Dsm** tab enables you to track program execution in the disassembly-level view
- the **Src** tab enables you to track program execution in the source-level view
- the file tab `dhry_1.c` shows the name of the current source file in the editing, or non-execution, view.

The **Src** tab acts like a button to display the source file that contains the line pointed to by the PC, if it is available. If the source is not available, RealView Debugger displays a No source message in the tab.

Click on the relevant tab to toggle between the different code views.

5.10.4 Viewing target status

The Processor status, in the Status line, shown in Figure 4-1 on page 4-3, enables you to see the current state of your debug target:

- | | |
|----------------|--|
| Unknown | Shows that the current state of the target is unknown to the debugger. For example it might have been running when the connection was established or it might be disconnected. |
| Stopped | Shows that the target is connected but any image loaded is not executing. |
| Running | Shows that an image is executing. In this case, a running progress indicator is also included. |

5.10.5 Using breakpoints in RealView Debugger

Breakpoints are specified locations where execution stops. A breakpoint is triggered either by:

- execution reaching the specified address
- data values at the specified location, in the current context, changing or becoming equal to a particular value or condition.

Breakpoints enable you to suspend program execution when the program accesses specific memory locations, variables, or functions. You can define conditions that are tested or qualify the breakpoint to define when execution stops. This section describes:

- *Breakpoint types* on page 5-31

- *Halted System Debug and Running System Debug* on page 5-32.

Breakpoint types

RealView Debugger enables you to use different types of breakpoint when you are debugging your image. RealView Debugger enables you to set software or hardware breakpoints:

Software These breakpoints are controlled by RealView Debugger directly. Because software breakpoints require that RealView Debugger change executable instructions, you can only set software breakpoints for code stored in RAM.

You can set many software breakpoints.

Hardware These breakpoints are controlled by the EmbeddedICE® module of your hardware. The complexity and number of breakpoints you can set depends on the hardware support provided by your debug target.

Note

RealView Debugger reserves one breakpoint unit for internal use and so this might not be available to you. You are warned if you try to set a hardware breakpoint when the limit is reached.

All ARM architecture-based processors provide basic hardware breakpoint support. This enables you to test address-specific data values.

Where advanced hardware support is available on your target processor, or as implemented by your simulator software (for example RVISS), you can set more complex hardware breakpoints. These breakpoints might be data-dependent or take advantage of range functionality. For example, some processors enable you to chain two breakpoints together, so that the break triggers only when the conditions of both breakpoints are met.

Check your hardware characteristics, and your vendor-supplied documentation, to determine the level of support available for hardware breakpoints. Also, see *Viewing your hardware breakpoint support* on page 5-33.

Note

For RVISS, only the RealView Connection Broker interface supports hardware breakpoints.

Software and hardware breakpoints are classified depending on the qualifiers that you assign to the breakpoint:

Unconditional

A breakpoint is unconditional when no condition qualifier is assigned to the breakpoint.

You can set an unconditional breakpoint that has no condition qualifiers and no actions assigned, and without explicitly choosing whether it is a software or hardware breakpoint. This is called a *default breakpoint*, and either RealView Debugger or your target hardware determines what kind of breakpoint is set. For a default breakpoint, the program always stops if execution reaches a specified location (see *Setting default breakpoints* on page 5-33).

You can assign one or more actions to an unconditional breakpoint. When the breakpoint is triggered, the specified action is performed. For more details, see the chapter that describes working with breakpoints in the *RealView Debugger v1.8 User Guide*.

Conditional

A breakpoint is conditional when a condition qualifier is assigned to the breakpoint that tests for a specific condition. The conditions that must be satisfied can be defined based on data values or on counters.

The program stops if execution reaches a specified location and one or more of the predefined conditions are met (see *Setting conditional breakpoints* on page 5-34).

By default, RealView Debugger always sets a software breakpoint in RAM. If you attempt to set a software breakpoint at a Flash or ROM address, RealView Debugger sets a hardware breakpoint instead, and warns you that this has happened:

Warning: Failed to set Software Break - used Hardware Break instead (read-only memory?).

Halted System Debug and Running System Debug

RealView Debugger supports different debugging modes:

Halted System Debug

Halted System Debug (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system.

HSD is always available as part of the RealView Debugger base product.

Running System Debug

Running System Debug (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system.

RSD is only available where supported by your debug target. You must also have one or more RealView Debugger RTOS extensions installed and are not provided as part of the base product.

Where RSD is supported, RealView Debugger enables you to switch seamlessly between RSD and HSD mode using Code window controls or CLI commands. For full details on using RTOS extensions and running in RSD mode, see the chapter that describes RTOS support in *RealView Debugger v1.8 Extensions User Guide*.

5.10.6 Viewing your hardware breakpoint support

To see the hardware breakpoint support for your debug target:

1. Select the following option from the Code window main menu:
Debug → Breakpoints → Hardware → Show Break Capabilities of HW...
 This displays an information box describing the support available for your target processor.
2. Click **Close** to close the information box.

5.10.7 Setting default breakpoints

A default breakpoint is the simplest breakpoint that you can set. A default breakpoint does not have any condition qualifiers or actions, and so is always an unconditional breakpoint. To set a default breakpoint:

1. Make sure line numbers are displayed (see *Displaying line numbers* on page 5-29).
 This is not necessary but might help you to follow the examples in this tutorial.
2. Right-click in the first entry in the Memory pane, <NoAddr>, and select **Set Start Address...** from the **Memory** context menu.
3. Enter a value as the start address for the area of interest, for example 0x8008.
4. Click **Set** to confirm the setting and close the dialog box.
5. Click on the **Src** tab in the File Editor pane.
6. Set an default breakpoint at line 149 in `dhry_1.c, Proc_5()`; . To do this double-click on the line number.

If the line number is not visible, then double-click inside the gray area at the left of the required statement in the File Editor pane to set the breakpoint.

Because the breakpoint location is in RAM, RealView Debugger sets an unconditional software breakpoint.

7. Set a watch on the variable `Int_1_Loc` at line 152 in `dhry_1.c`. To do this right-click on the variable. The variable is then underlined in red.
8. Select **Watch** from the context menu.
9. Click the **Run** button to start execution.
10. Enter the required number of runs, for example `20000`.
11. Monitor execution until the breakpoint is reached.
12. Click the **Run** button again and monitor the program as execution continues.
13. Double-click on the red breakpoint indicator to clear the breakpoint.

See the chapter that describes working with breakpoints in the *RealView Debugger v1.8 User Guide* for more details.

5.10.8 Setting conditional breakpoints

Use the Set Address/Data Breakpoint dialog box to set a conditional breakpoint:

1. Select **Target** → **Reload Image to Target** to reload the image.
2. Select **File** → **Open...** and open the source file `dhry_2.c` so that it is displayed in the File Editor pane.
3. Scroll through the source file `dhry_2.c` and locate the line:

```
81 }/* Proc_7 */
```
4. Right-click on the line number and select **Set Break...** from the context menu to display the Set Address/Data Breakpoint dialog box shown in Figure 5-5 on page 5-35.

If the line number is not visible, then right-click inside the gray area at the left of the required statement or on the line of code to display the context menu.

————— Note —————

Ensure that you select **Set Break...** from the context menu. If you select **Set Break**, the Set Address/Data Breakpoint dialog box is not displayed and the breakpoint is set automatically.

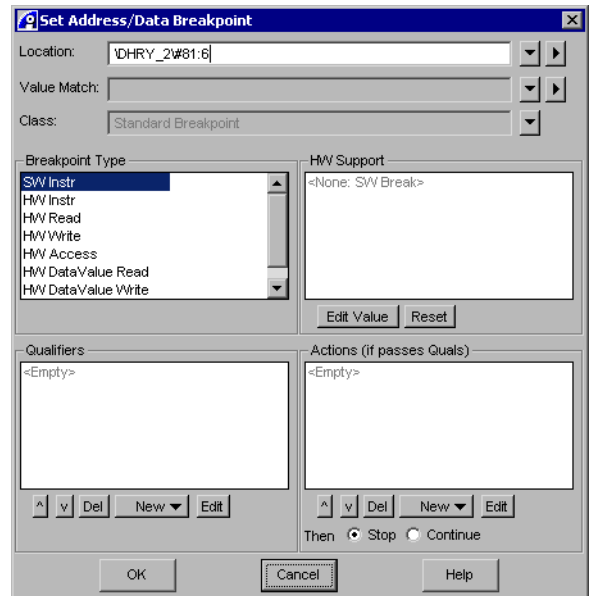


Figure 5-5 Set Address/Data Breakpoint dialog box

In this dialog box:

- The Location field is prefilled (but you can change this if required).
- The Class field shows the default breakpoint supported by your debug target (a Standard breakpoint). You cannot change this.

The contents of this field change depending on your debugging environment and target configuration, see *RealView Debugger v1.8 User Guide* for details.

- By default, a software instruction breakpoint is highlighted. The available breakpoint types are defined by your debug target. For connections that use RVISS, you can choose from a range of supported breakpoint types.
 - For certain breakpoint types, the Location, Value Match or Class fields might be unavailable. In this case, the field is grayed out and right-click menus are disabled.
5. Modify the Location field to the value `\DHRV_2\#81:6`, as shown in Figure 5-5.
 6. Click **New** to specify the Qualifiers or conditions for the new breakpoint. This displays the **New Qualifiers** menu.
 7. Select **When Expression True...** from the menu.
 8. Enter the condition `Int_Glob==5` in the prompt.

RealView Debugger requires that you enter the condition in C format.

9. Click **Set** to confirm the entry and close the prompt box.

The breakpoint condition is shown in the Qualifiers group display.

10. Click **OK** to close the Set Address/Data Breakpoint dialog box.

The code view is updated and a yellow disc appears to show that you have set a conditional breakpoint. The **Cmd** tab, in the Output pane, shows the RealView Debugger command used to set the breakpoint.

Because the breakpoint location is in RAM, RealView Debugger sets a conditional software breakpoint.

11. Click the **Run** button on the toolbar to execute the image, and enter 20000 at the prompt.

When the breakpoint is reached, execution stops and RealView Debugger displays a message in the **Cmd** tab of the Output pane to identify the point in the source code where the program stopped.

The next section, *Managing breakpoints*, describes how to display the breakpoint details, and other useful tasks.

If you try to set a breakpoint on a non-executable line, RealView Debugger looks for the first executable line immediately following and places the breakpoint there. If the lines preceding the breakpointed instruction are comments, declarations, or other non-executable code, they are marked with gray, downward pointing arrows. Lines marked in this way are regarded as part of the breakpoint. You cannot place two unconditional breakpoints on the same line, or on lines marked by the downward pointing arrows.

For full details on using the Set Address/Data Breakpoint dialog box to specify your breakpoints, see the chapter that describes working with breakpoints in *RealView Debugger v1.8 User Guide*.

5.10.9 Managing breakpoints

To display the current breakpoint details:

1. Select **View** → **Break/Tracepoints** from the default Code window to display the Break/Tracepoints pane, shown in Figure 5-6 on page 5-37.

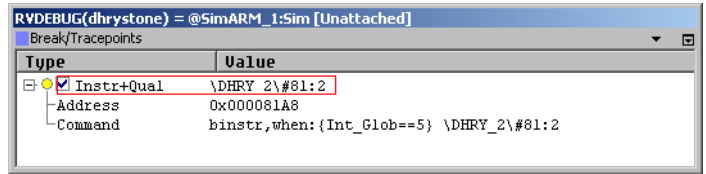


Figure 5-6 Break/Tracepoints pane

This shows the conditional breakpoint you set previously, the address, and the RealView Debugger command used to create it. RealView Debugger has identified the appropriate class of breakpoint, based on your debug target.

The check box is selected to show that the breakpoint is enabled.

2. Right-click anywhere on the breakpoint entry, in the Break/Tracepoints pane, to display the **Break/Tracepoints** menu.

From here, you can examine the breakpoint, edit or copy it, or disable it so that it is ignored the next time the program runs.

3. To remove the current breakpoint, right-click on the conditional breakpoint entry and select **Clear** from the **Break/Tracepoints** menu.

The breakpoints list is refreshed after the entry has been removed.

4. Click the **Run** button again to complete program execution.

Using CLI commands

You can use CLI commands to create and modify breakpoints. For example, to set the conditional breakpoint shown in Figure 5-6, type:

```
binstr,when:{Int_Glob==5} \DHR_2\#81:6
```

It is not necessary to specify the breakpoint class.

For full details see the BREAK commands in *RealView Debugger v1.8 Command Line Reference Guide*.

5.10.10 Setting watches

Watches monitor variables during execution of your image. To set a series of watches and to monitor their value during execution of the image:

1. Select **Target** → **Reload Image to Target** to reload the image.
2. At line 127 in dhry_1.c, right-click on the line number and select **Set Break** from the context menu, to set an unconditional breakpoint.

Note

This is not the same as the **Set Break...** option.

3. On line 125, right-click on the variable `Number_Of_Runs` to display the **Source Variable Name** context menu.
4. Select **Watch** from the menu. This adds the chosen variable to the Watch pane and displays the current value, if known. The Watch pane is the center pane in the middle row of the default Code window (see Figure 4-1 on page 4-3).
5. Click the **Run** button on the Debug toolbar to execute the image. Enter a large number of runs, for example 50000.

The program stops execution at line 127, and the Watch pane shows the current value in `Number_Of_Runs`. It is shown as hexadecimal by default. To see the value as an unsigned decimal:

- a. Right-click on the variable in the Watch pane, and select **Format...** from the context menu.
- b. In the List Selection dialog box, select **Unsigned Decimal**.
- c. Click **Ok**.

The value changes to an unsigned decimal.

6. Click the **Run** button again on the toolbar to execute the image.
7. Click the **Stop Execution** button to stop the program before it finishes.
With the processor halted, monitor changes in the variables in the Watch pane. Depending on where you halted execution, values that changed at the last pane update are displayed in dark blue. Values that changed at a previous pane update are displayed in light blue.
8. Click the **Run** button again to complete program execution.
9. Clear the breakpoint on line 127 in `dhry_1.c`.
10. Remove the variables from the Watch pane as follows:
 - a. Click the name of the variable in the Watch pane to select it.
 - b. Press Delete.

The values list is refreshed after the entry has been removed.

For full details on setting watches, see the chapter that describes working with debug views in *RealView Debugger v1.8 User Guide*.

5.10.11 Displaying register contents

To display register contents for the loaded image:

1. Switch the Process Control pane to the Register pane. To do this:

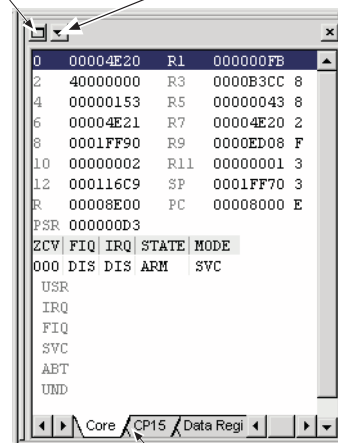


- a. Click the **Pane Control** menu button.
- b. Select **New Pane** → **Register** from the **Pane Control** menu.

The Process Control pane switches to the Register pane. It looks like the example in Figure 5-7.

Pane Content menu,
click here to change
the pane view

Pane Menu,
click here to work
with pane data



Select a tab to view target details

Figure 5-7 Register pane

The Register pane displays tabs appropriate to the target processor running your image and the target vehicle used to make the connection. For the ARM7TDMI core using RVISS, the pane includes the **Core** tab, showing the base registers for the connected target processor, and the **CycleCount** tab, showing internal debugger variables and statistics.

If you are connected to real target hardware, for example using RealView ICE, the Register pane might include other tabs, for example **CP15** and **Debug**.

2. Click the **Pane** menu and select **Show Enumerations as Values** from the available options. This displays the register contents as values rather than enumerated strings. The Register pane is refreshed.

3. Click the **Pane** menu and unselect **Show Enumerations as Values**.

5.10.12 Changing register contents

To change the contents of registers:

1. Click on the **Core** tab in the Register pane.
2. Right-click in the Mode field value of the *Current Program Status Register* (CPSR) register (SVC) to display the **Register** context menu.
3. Select **Set Enumeration...**. This opens a selection box where you can specify the context-sensitive value to modify the selected register entry.
4. Select FIQ from the list of available options and click **OK** to change the Mode field of the CPSR from SVC (0013) to FIQ (0011). This updates the contents of the ARM banked registers, shown in the pane. Similarly, if you are in SVC mode and you change the contents of R13 in the SVC bank (SP), the contents of R13 (SP) change to match.

In this example, the value can also be changed by selecting **FIQ** from the list of values from the context menu.

For full details on working in the Register pane, and the contents of tabs, see the chapter that describes working with debug views in *RealView Debugger v1.8 User Guide*.

5.10.13 Process control

To display the Process Control pane:

1. Switch the Register pane back to the Process Control pane. To do this:
 - a. Click the **Pane Control** menu button.

The name of the previous pane (Process Control) is included at the top of the menu. This is an alternative to selecting the pane from the **New Pane** menu option.
 - b. Select **Process Control** from the **Pane Control** menu.

The Register pane switches back to the Process Control pane. It looks like the example in Figure 5-8 on page 5-41.

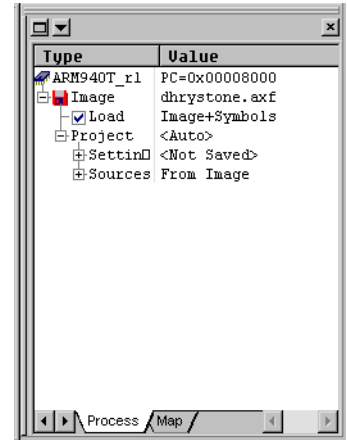


Figure 5-8 Process Control pane

The **Process** tab shows details about your current process. If you are debugging a single process application, this is the same as viewing the processor details. In this example, you can identify the target processor and see details about your project and the loaded image.

2. Right-click on an entry in this pane, for example `dhrystone.axf`, and select **Properties** from the context menu. This displays a text box giving more information on the chosen item.

Working with source files

When working with entries in the Process Control pane, you can use type ahead facilities to locate files. This is especially useful where your project specifies a large number of source files. For example, type the first letter of the source file that you want to view. RealView Debugger expands the Sources entry and locates the first matching occurrence. When using this feature, the type ahead buffer is case insensitive and is limited to 128 characters. Do one of the following to clear the buffer:

- select a different item
- press Home to move to the top of the pane
- press Escape.

Working with images

Where you have created a user-defined project, it is recommended that you open this first to load and debug the associated image, or images. This enables you to access the project properties, save new settings, or make changes to the build model. In this example, the user-defined project is open so project settings have been used to populate entries in the **Process** tab.

When an image is loaded directly to a debug target, RealView Debugger checks to see if an auto-project exists for that image in the same location. Where no auto-project exists, RealView Debugger creates an in-memory project to use in the current session.

An auto-project is a custom, image control, project that holds project settings where the build model is unknown. You can view these settings using the Project Properties window. Using an auto-project enables you to amend and save image load parameters where you do not have a user-defined project or where you have no control over the build model. You also have the option to use the saved auto-project as the basis of a user-defined project.

Note

If you load an image built as part of a user-defined project without opening the project you cannot access all the project properties because these are unknown to RealView Debugger. In this case, RealView Debugger creates an in-memory project, or uses the saved auto-project file.

See the chapter that describes working with images in *RealView Debugger v1.8 User Guide* for details on using auto-projects in RealView Debugger.

5.10.14 Displaying memory contents

The Memory pane shows the contents of memory on your debug target. In the default Code window, the Memory pane is the right-hand pane of the middle row (see Figure 4-1 on page 4-3).

To display an area of memory:

1. Select **Target** → **Reload Image to Target** to reload the image.
2. Right-click on the address column in the Memory pane to display the **Memory** context menu.
3. Select **Set Start Address...** to display the address prompt dialog box.
4. Enter 0x8000 as the new start address

5. Click **Set** to confirm set the start address for the memory pane and close the prompt dialog box. The Memory pane changes to show the memory contents starting at 0x8000.

You might see some memory contents shown in dark blue. These are memory locations where the value has changed.



6. Click on the **Pane** menu of the Memory pane.
7. Select **Format** → **Show ASCII** to include the ASCII equivalent of the memory contents, shown in Figure 5-9.

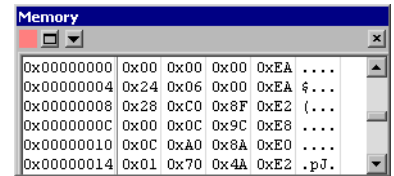


Figure 5-9 Updated memory contents

8. Click on the **Pane** menu again.
9. Select **Set Number of Columns to show...** to choose how many columns are used in the memory display, for example 8. If you do not specify the number used (or specify zero), RealView Debugger displays as many columns as it can fit into the pane.
10. Click the **Run** button on the Debug toolbar to execute the image.
11. Enter a large number of runs, for example 50000.
12. Click **Stop Execution** to stop the program before it finishes and view the updated memory contents.
Any memory contents that are colored light blue are those that have changed since the previous update, and depends on where execution stops.
13. In the source file that has the focus, double-click on a variable or function name so that it is highlighted and then drag it to the Memory pane where you can drop it into the display. This is a quick way to display a chosen location in the Memory pane.

Use the **Pane** menu, or right-click in an address entry in the Memory pane, to select the display format and modify the address range of the memory area that you want to see.

You can change memory contents displayed in the Memory pane using in-place editing.

5.10.15 Displaying variables

To display the value of a variable from your source code:

1. Select **Target** → **Reload Image to Target** to reload the image.
2. Double-click in the margin of line 301 in `dhry_1.c` to set a default breakpoint.
3. Click the **Run** button on the toolbar to execute the image.
4. Enter the required number of runs when prompted, for example `20000`.
5. When execution stops at line 301, highlight the variable `Ptr_Glob` in the expression:

```
structassign (*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
```
6. Right-click to display the **Source Variable Name** context menu.
7. Select **Print** to view the value of the chosen variable in the current context. This is displayed in the **Cmd** tab of the Output pane.
8. Right-click on the variable again to display the **Source Variable Name** context menu, then select **View Memory At Value**. A floating Memory pane is displayed showing the memory at the location of the variable.

5.10.16 Tooltip evaluation

Use **Tooltip Evaluation** to see hover-style evaluation when you hold your mouse pointer, for two seconds, over a variable in the **Src** tab, or a register in the **Dsm** tab.

This option is enabled by default. Select **Edit** → **Advanced** → **Tooltip Evaluation** to disable this for the current Code window.

5.10.17 Using the call stack

The Call Stack pane enables you to follow the flow of your program by examining the current status of functions and variables. This pane shows you the path that leads from the main entry point to the currently executing function. In the default Code window, the Call Stack pane is the left-hand pane in the middle row (see Figure 4-1 on page 4-3).

To monitor your execution path:

1. Select **Target** → **Reload Image to Target** from the Code window main menu to reload the image.
2. Click the **Run** button on the Debug toolbar to execute the image. Enter a large number of runs, for example `50000`.

3. Click **Stop Execution** to stop the program before it finishes.
The **Call Stack** tab, in the Call Stack pane, displays details of the functions currently on the stack and awaiting execution.
4. Right-click on an entry in the Call Stack pane, to see the **Function** context menu.
This menu enables you to carry out operations on the chosen function in the stack, for example to scope to that function.
5. Click on the **Locals** tab in the Call Stack pane. This displays a list of the variables that are local to the current function, shown in the example in Figure 5-10.

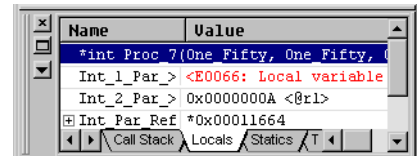


Figure 5-10 Local variables in the Call Stack pane

If a variable is a structure or an array, a plus sign is added to the entry in the Call Stack pane. You can click on this to expand the variable to see all elements of the structure or array.

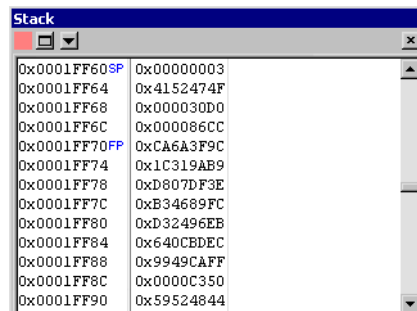
6. Right-click on an entry in the **Locals** tab, to see the **Variables** context menu.
7. Click on the **Statics** tab, to see non-local variables, that is module statics.

5.10.18 Using the stack

The Stack pane enables you to monitor the contents of the stack as raw memory and to make changes to those settings. The Stack pane enables you to follow the flow of your application through the hierarchical structure and shows you the path that leads from the main entry point to the currently executing function.

To view the Stack pane:

1. Select **View** → **Stack** from the default Code window. The Stack pane is displayed as a floating pane.
2. Select **Target** → **Reload Image to Target** from the Code window main menu to reload the image.
3. Click the **Run** button on the Debug toolbar to execute the image. Enter a large number of runs at the application prompt, for example 50000.
4. Click **Stop Execution** to stop the program before it finishes.
5. View the updated Stack pane, shown in Figure 5-11 on page 5-46.



Address	Value
0x0001FF60	SP
0x0001FF64	0x00000003
0x0001FF68	0x4152474F
0x0001FF6C	0x000030D0
0x0001FF70	FP
0x0001FF74	0x000086CC
0x0001FF78	0xCA6A3F9C
0x0001FF7C	0x1C319AB9
0x0001FF80	0xD807DF3E
0x0001FF84	0xB34689FC
0x0001FF88	0xD32496EB
0x0001FF8C	0x640CBDEC
0x0001FF90	0x9949CAFF
0x0001FF94	0x0000C350
0x0001FF98	0x59524844

Figure 5-11 Viewing the stack

The stack pointer, marked by SP, is located at the bottom of the stack. The frame pointer, marked by FP, shows the starting point for the storage of local variables.

Changing the stack pointer

As you step through your code, the default stack pointer is used, shown in Figure 5-11. You can specify an expression or a register to use as the stack pointer from the Stack pane:

1. Right-click in the left column of an entry to display the **Stack** context menu.
2. Select **Set Start Address...** to display the address prompt box.
3. Enter an expression or a register, for example @R9, as the new stack pointer.
4. Click **Set** to confirm your choice and close the address prompt box.

The new stack pointer is marked by *Expression Pointer* (EP), located at the bottom of the stack.

If you enter a blank expression, or remove the existing expression, in the address prompt box, RealView Debugger reverts to using the default stack pointer register. In this example, this was R13 shown in Figure 5-7 on page 5-39.

5. Select **Hide** from the **Pane Control** menu to hide the Stack pane.

5.10.19 Using the Data Navigator pane

RealView Debugger provides a Data Navigator pane to help with debugging tasks. This pane enable you to search through your source files to look for specific structures and to monitor their status during program execution. The pane enables you to search:

- images (especially useful if you have a multi-image application)
- modules
- functions

- variables.

To access the Data Navigator pane, select **View** → **Data Navigator** from the Code window main menu. The **Images** tab is selected, which lists the images that you have loaded to the current debug target.

To locate a specific module, function, or variable in an image:

1. Double-click on the required image name (for example, @dhrystone) to narrow the list of modules, functions, and variables to that image. The **Modules** tab is then selected, which lists the modules for the chosen image.
The dhrystone example is a single image application. If you have a multi-image application, you can narrow the search to a specific image.
2. Locate the module containing the required function, for example DHR_1.
3. Double-click on the module name. The **Functions** tab is displayed, which lists the functions in the chosen module.
4. Double-click on the function name, for example Proc_2. RealView Debugger displays the disassembly view in the **Dsm** tab of the Output pane at the start of the chosen function.
5. Alternatively, to locate the function in your source code:
 - a. Right-click on the required function entry (for example, Proc_3) to display the **Functions** context menu.
 - b. Select **Show Source**.
RealView Debugger locates the function in the source code tab of the file editor pane. If the source file containing the function is not already open, RealView Debugger first opens the source file.
6. Select **Hide** from the **Pane Control** menu to hide the Data Navigator pane.

See the chapter that describes working with browsers in *RealView Debugger v1.8 User Guide* for full details on using the Data Navigator pane.

5.10.20 Using the Symbol Browser pane

RealView Debugger provides a Symbol Browser pane for viewing C++ classes. To access the Symbol Browser pane, select **View** → **Symbol Browser** from the Code window main menu. See the chapter that describes working with browsers in *RealView Debugger v1.8 User Guide* for full details on using the Symbol Browser pane.

5.11 Unloading an image

RealView Debugger automatically unloads an image from a debug target when you:

- disconnect from the debug target
- exit RealView Debugger, and you choose to disconnect the connection.

Also, if you close the RealView Debugger project associated with an image, RealView Debugger displays a prompt asking if you want the image unloaded.

However, you might want to unload an image explicitly as part of your debugging session, for example if you correct coding errors and then rebuild outside RealView Debugger. See *How to explicitly unload an image* for instructions.

You do not have to unload an image from a debug target before loading a new image for execution. Display the Load File to Target dialog box and ensure that the **Replace Existing File(s)** check box is selected ready to load the next image (see *Loading an image directly* on page 5-19).

5.11.1 How to explicitly unload an image

You can unload an image by using the Process Control pane. To do this:

1. If the Process Control pane is hidden, select **View** → **Process Control** from the default Code window main menu.

———— **Note** ————

If you have still have the default panes visible in the Code window, as shown in Figure 4-1 on page 4-3, then you do not have to do this step.

2. Right-click on either the Image or Load entry to display the **Image** context menu.
3. Select **Unload**.

Alternatively, in the Process Control pane click on the check box associated with the Load entry so that it is not selected.

———— **Note** ————

Unloading an image does not affect target memory. It unloads the symbols and removes most of the image details from RealView Debugger. However, the image name is retained.

To remove image details completely, right-click on either the Image or Load entry in the Process Control pane and select **Delete Entry** from the **Image** context menu.

5.12 Saving the session

RealView Debugger stores session details by default when you end your debugging session. Saving the session enables you to start your next session using the same working environment, connecting automatically to a specified target, and opening projects.

RealView Debugger stores session details using your:

- *Workspace*
- *Startup file* on page 5-50
- *History file* on page 5-50
- *Settings options* on page 5-51.

5.12.1 Workspace

The RealView Debugger workspace is used for visualization and control of default values, and storing persistence information. It includes:

- user-defined options and settings
- connection details
- details about open windows and, in some cases, their contents
- user-defined projects or auto-projects to open on startup.

The first time you run RealView Debugger, the default workspace settings file `rvdebug.aws` is created in your home directory. Each time you start RealView Debugger after this, the debugger loads this workspace automatically, but you can change the defaults or create a new workspace of your own.

Making changes to the workspace

To make changes to your workspace:

1. Select **File** → **Workspace** → **Workspace Options...** from the main menu to display the workspace Settings Window.

2. Make the changes you require.

For full details on amending your workspace settings to define how session details are saved and where, see the chapter that describes configuring workspaces in *RealView Debugger v1.8 User Guide*.

3. When you have made the required changes, select **File** → **Save and Close**.

Closing your workspace

To close your workspace so that RealView Debugger runs without a workspace, Select **File** → **Workspace** → **Close Workspace**.

Starting RealView Debugger without a workspace

You can start RealView Debugger from the command line without loading a workspace, for example:

```
rvdebug.exe -aws=-
```

5.12.2 Startup file

The startup file contains a record of your last debugging session including:

- images and files loaded into RealView Debugger
- the list of all recently loaded files, for example source files
- the recent workspaces list
- the workspace to be used on startup, if specified
- workspace save and restart settings
- user-defined menu settings, for example pane format options
- the recent projects list.

By default, this file is called `rvdebug.sav`. The first time you exit RealView Debugger after performing an operation in the default Code window, a startup file is created in your home directory. From this point on, every time you close down RealView Debugger and exit, this startup file is updated. You can specify a different startup file, or none at all, by changing your workspace settings.

5.12.3 History file

The history file contains a record of:

- Commands submitted during a debugging session, for example, changing directory, loading source files, loading an image for execution, or setting breakpoints.

Breakpoints are only added to the history list if they are set using breakpoint dialog boxes, for example the Set Address/Data Breakpoint or the Simple Break if X dialog box. If you set a breakpoint in another way, for example using a CLI command, this is not added to the list. See the chapter that describes working with breakpoints in *RealView Debugger v1.8 User Guide* for more details.

- Data entries examined in the Code window during debugging.

- The Set Directory and Set File lists used in Open dialog boxes, for example Select File to Open, or Select File to Include Commands from.
- Up to 32 personal favorites such as variables, data values, watches, and breakpoints.

The first time you run RealView Debugger, carry out an operation (for example open a file or load an image) and then exit, a history file is created in your home directory. By default, this is called `exphist.sav` and is updated at the end of all subsequent debugging sessions.

Note

If you are using RealView Debugger on non-Windows platforms, the history file is only created if you create and save a favorite, for example a breakpoint or watchpoint. See Appendix B *RealView Debugger for Sun Solaris and Red Hat Linux* in *RealView Debugger v1.8 User Guide* for details.

5.12.4 Settings options

By default, RealView Debugger:

- writes your current settings to your startup file ready for re-use
- saves your current workspace
- reloads this workspace at the next start-up.

To change these defaults, select **File** → **Workspace** from the main menu to display the **Workspace** menu. At the bottom of the **Workspace** menu are your current settings options. Enable or disable the required options to decide how your next RealView Debugger session starts:

Save Settings on Exit

If unselected, any changes you make to your workspace are lost when you exit RealView Debugger. The next time you start RealView Debugger, it opens either the previous saved workspace settings if **Same Workspace on Startup** is selected, or the default workspace.

Same Workspace on Startup

If unselected, RealView Debugger does not use the current workspace at the next start-up. This means that, unless another workspace is specified on the command line, RealView Debugger opens the default workspace.

The state of these options are saved in your startup file when you exit RealView Debugger.

5.13 Disconnecting from a target

You can disconnect from the current target in one of the following ways:

- Select **Target** → **Disconnect** from the Code window main menu.
This immediately disconnects the current connection.
-  Use the **Disconnect** button on the Connect toolbar.
This immediately disconnects the current connection.
- Select **Target** → **Connect to Target...** from the Code window main menu to display the Connection Control window, and unselect the check box to disconnect the connection.



You can also click the **Connection Control** button on the Connect toolbar to display the Connection Control window.

RealView Debugger Code windows do not close when you disconnect from a target. However, if you have an image loaded, disconnecting removes all the debug information from RealView Debugger and this clears pane contents.

Disconnecting changes the Processor status to *Unknown*, and the CLI prompt changes to *None>*.

Disconnecting does not close any open projects.

———— Note ————

It is not necessary to disconnect from a target before you close down RealView Debugger. See *Exiting RealView Debugger* on page 5-54 for more details.

5.13.1 Setting disconnect mode

You can control the way a target processor is left after you disconnect. This is particularly useful when debugging multiprocessor debug targets and working with multiple threads. In single processor debugging mode, you might want to leave an image executing but close down RealView Debugger.

If you are connected, you can set the disconnect mode. To do this:

1. Select **Target** → **Disconnect (Defining Mode)...**
2. Select the required state from the Disconnect Mode selection box.

The options listed depend on your target vehicle, for example, when connected using RealView ARMulator ISS, the only disconnect option available is **As-is with Debug**.

3. Click **OK** to disconnect and leave the processor in the required state.

Note

If you click **Cancel**, the disconnection continues using the default disconnect mode as defined by the target vehicle.

If you set disconnect mode from the Connection Control window, this temporarily overrides any setting in your target configuration file.

See the chapter that describes connecting (and disconnecting) in *RealView Debugger v1.8 Target Configuration Guide* for full details of disconnect mode and how to specify this setting for your debug target.

Setting disconnect mode for multiprocessor debugging

If you are working in multiprocessor debugging mode you can specify a disconnect mode for any of your active connections:

1. Select **Target → Connect to Target...**, or click the **Connection Control** button, to display the Connection Control window.
2. Right-click on the connection entry and select **Disconnect (Defining Mode)...** from the **Disconnection** context menu.
3. Select the required state from the selection box.
The options listed depend on your target vehicle.
4. Click **OK** to disconnect and leave the processor in the required state.

Note

If you click **Cancel**, the disconnection continues using the default disconnect mode as defined by the target vehicle.

5.14 Exiting RealView Debugger

This section describes the options available when you exit RealView Debugger:

- *Closing down without a connection*
- *Closing down with a connection* on page 5-55
- *Reconnecting to a target* on page 5-56.

If you chose to save the current state of your Code window and connection when you exit RealView Debugger (this is the default, but see *Settings options* on page 5-51), then next time you start RealView Debugger:

- the Code window appears in the same state as your previous debugging session
- RealView Debugger automatically attempts to reconnect to a debug target, if it was previously connected when you last exited RealView Debugger
- RealView Debugger automatically opens any project that was open when you last exited RealView Debugger.

5.14.1 Closing down without a connection

Disconnect from your target, as described in *Disconnecting from a target* on page 5-52, and select **File** → **Exit** to display the Exit dialog box.

Click **Yes** to close the Exit dialog box and close down RealView Debugger.

If you have made any RealView Connection Broker connections during the debugging session, for example to RVISS, and you have disconnected all the connections before you exit, then RealView Debugger leaves the RealView Connection Broker running. When you next start RealView Debugger, further RealView Connection Broker connections are numbered sequentially from the last highest connection number, plus one. For example:

1. Start RealView Debugger.
2. Make three connections to RVISS using the new_arm connection. The connections are numbered simarm_1, simarm_2, and simarm_3.
3. Disconnect all connections, then exit RealView Debugger. Click **Yes** when prompted to exit.
4. Start RealView Debugger again.
5. Make another connection to RVISS. This connection is numbered simarm_4.

However, if you shut down RealView Connection Broker after exiting RealView Debugger, then the next time you start RealView Debugger the numbering of RVISS connections starts at one.

5.14.2 Closing down with a connection

If you are connected to your target and you select **File** → **Exit**, the Exit dialog box includes the **Disconnect Connection** check box, shown in Figure 5-12.

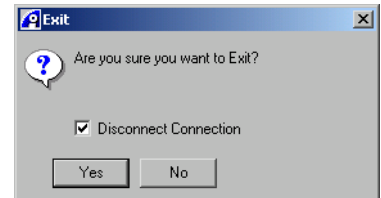


Figure 5-12 Exit dialog box, connected

Use the check box to specify your disconnect options:

- *Maintaining the connection*
- *Terminating the connection.*

Maintaining the connection

To close down and maintain the connection:

1. Unselect the **Disconnect Connection** check box shown in Figure 5-12.
2. Click **Yes** to close the Exit dialog box and close down RealView Debugger.

RealView Debugger exits but the connection is maintained for future sessions, that is the *Target Vehicle Server* (TVS) is left running. If you are connected to a remote debug target and you close down this way, the remote RealView Connection Broker, that is RealView Network Broker, is also left running.

Terminating the connection

To close down and terminate the connection:

1. Leave the **Disconnect Connection** check box, shown in Figure 5-12, so that it is selected.
2. Click **Yes** to close the Exit dialog box and close down RealView Debugger.

RealView Debugger exits and terminates any connections to debug targets, that is TVS stops. If you are connected to a remote debug target and you close down this way, you must stop the remote RealView Connection Broker, that is RealView Network Broker, manually.

Note

See *Components of RealView Debugger* on page 3-4 for details of TVS and RealView Connection Broker.

5.14.3 Reconnecting to a target

RealView Debugger saves all open connections in the current workspace if you close down without disconnecting first. This default behavior is independent of the status of the **Disconnect Connection** check box shown in Figure 5-12 on page 5-55.

This means that, if you exit a session when still connected to a debug target, RealView Debugger tries to reconnect when you next open the workspace. However, this might fail if the connection or vehicle status has changed, for example if your RealView ICE interface has been disconnected or if your Multi-ICE® server has stopped.

If RealView Debugger reconnects successfully, the connection mode specified in the target configuration file is used by default. See the chapter that describes connecting in *RealView Debugger v1.8 Target Configuration Guide* for full details on defining the connect mode for your debug target.

Note

Use your settings options to change the way RealView Debugger closes down, see *Settings options* on page 5-51 for details.

5.15 Completing the tutorial

If you created the tutorial project directory described in *Starting the tutorial* on page 5-3 you might want to remove it. Because the tutorial project is self-contained, follow the steps described in the following sections:

- *Removing tutorial project files*
- *Removing the tutorial project.*

However, if you intend to follow the tasks described in the following chapters, complete these tasks before removing the tutorial project:

- Chapter 6 *Getting Started with RealView Debugger Projects*
- Chapter 7 *Getting Started with the RealView Debugger Command Line Interface.*

5.15.1 Removing tutorial project files

To remove the files associated with the project:

1. Delete the project file:
`install_directory\RVDS\Examples\...\Tutorial\dhystone.prj`
2. Delete the build target configuration directories, for example:
`install_directory\RVDS\Examples\...\Tutorial\Debug`
3. Delete the target generated makefiles, for example:
`install_directory\RVDS\Examples\...\Tutorial\dhystone_Debug.mk`
4. Delete any backup files with the .bak extension.

The project is now deleted from your workstation.

5.15.2 Removing the tutorial project

If you do not want to keep the files copied at the start, or the executable image, you can delete the entire tutorial project base directory:

`install_directory\RVDS\Examples\...\Tutorial.`

Chapter 6

Getting Started with RealView Debugger Projects

This chapter introduces you to RealView® Debugger projects. It contains the following sections:

- *Help on creating RealView Debugger projects* on page 6-2
- *Setting up project build options for the basic compilation tasks* on page 6-8.

For complete instructions on how to create and manage RealView Debugger projects, see the *RealView Debugger v1.8 Project Management User Guide*.

6.1 Help on creating RealView Debugger projects

This section helps you to determine what kind of RealView Debugger project to create. It contains the following sections:

- *Types of project*
- *Project properties* on page 6-3
- *Limitations of Standard and Library projects* on page 6-5
- *Making a project and a connection interdependent* on page 6-6
- *Automatic operations performed by a project* on page 6-6.

6.1.1 Types of project

In RealView Debugger, you can create the following types of project:

- | | |
|------------------|---|
| Standard | Builds an executable image using the compiler, assembler and linker as appropriate. You add the sources used to build the image to the project, and RealView Debugger creates the necessary makefiles for you. However, there are limitations with this type of project (see <i>Limitations of Standard and Library projects</i> on page 6-5). |
| Library | Builds an object library using the compiler, assembler, and the ARM® librarian utility as appropriate. You add the sources used to build the library to the project, and RealView Debugger creates the necessary makefiles for you. However, there are limitations with this type of project (see <i>Limitations of Standard and Library projects</i> on page 6-5). |
| Custom | Builds any build target (executable image, ROM image, or object library) from your own makefile. You specify the location of your makefile, and any arguments that it takes. This type of project also enables you to overcome the limitations of the Standard and Library projects (see <i>Limitations of Standard and Library projects</i> on page 6-5). |
| Container | Builds your application from subprojects you have previously created. You can add any combination of Standard, Library, and Custom subprojects. A Container project can also include other Container projects, but must not include itself. However, the order in which you add subprojects determines the build order. |

6.1.2 Project properties

When you create a project, RealView Debugger sets up default project settings, called Project Properties. These settings include project information and build options that you can modify. The project properties are stored in project files that have the .prj file extension. A project file is located in the directory you specify when you create a project.

You can access the Project Properties for your project using a Project Properties window. For more details on accessing the Project Properties, and how to modify them, see the chapter that describes managing projects in the *RealView Debugger v1.8 Project Management User Guide*.

For more information about the compilation tool options used to implement the project build options, see *Setting up project build options for the basic compilation tasks* on page 6-8 and the compilation tools documentation.

Build target configurations for Standard and Library projects

When you create a Standard or Library project, RealView Debugger automatically opens a Project Properties window, and sets up default build target configurations, Debug, DebugRel, and Release. You can delete these build target configurations and create your own as required. However, for the purposes of this tutorial, it is suggested that you use the default configurations.

The build-related groups also include special child groups corresponding to each of the build target configurations. These are identified by a key symbol and the configuration name (see Figure 6-1).

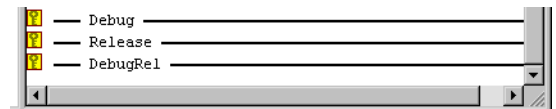


Figure 6-1 Build target configuration child groups

Each build target configuration can have different values for the build settings, but only one configuration is active for a project. For example, you might want to set up and use the Debug and Release configurations to build images for debugging and final release.

Any setting that is not in one of the child groups is a global settings that is used for all build target configurations. You can move or copy settings to any of these child groups:

- If you move a global setting to a build target configuration group, the current value for that setting is used only for that build target configuration. Any other build target configurations use the default value for the setting.

- If you copy a global setting to a build target configuration group, you can change the copied setting to a configuration-specific value. Any other build target configurations use the global value for the setting.

The first time you close the Project Properties window after creating a Standard or Library project, RealView Debugger creates a build directory and makefile for each build target configuration defined for the project.

Note

If a makefile and directory for the active build target configuration does not already exist when you build a project, RealView Debugger displays a prompt stating that the makefile does not exist. Click **Yes** at the prompt dialog. RealView Debugger creates the makefile and directory for the active build target configuration, then performs the build.

See the chapter that describes managing projects in the *RealView Debugger v1.8 Project Management User Guide* for more details on build target configurations.

Preprocessing and post-processing commands

Standard and Library projects also enable you to specify other preprocessing and post-processing commands. For example, after RealView Debugger has built an image, you can set up a `fromelf` command that the project build process is to use to convert the image to a binary ROM image.

6.1.3 Limitations of Standard and Library projects

Table 6-1 describes the limitations with Standard and Library projects, and the consequences. To overcome these limitations, create your own makefile, and create a Custom project to use that makefile.

Table 6-1 Limitations of Standard and Library projects

Limitation	Consequence
Each source file you add is built with a separate assembler or compiler command.	You cannot have multiple source files in a single assembler or compiler command.
By default, all files of the same language type (C, C++, or assembler, and the ARM and Thumb® instruction set variants) are built using the same assembler or compiler options.	You cannot, by default, specify different assembler or compiler options for different source files of the same language type and instruction set variant. For example, if you have multiple source files written in C targeted at the ARM instruction set, they are all built using the same compiler options. ———— Note ———— You can create additional COMPILE or ASSEMBLE groups, each with different options, and place your source files in the appropriate groups. To do this, display the Project Properties for the project, right-click on the required group, and select Make New... or Make Copy... from the context menu. Give your new group a unique name, and set up the options as required. See <i>RealView Debugger v1.8 Project Management User Guide</i> for more details editing Project Properties.

6.1.4 Making a project and a connection interdependent

RealView Debugger provides a mechanism that enables a project and a connection to be mutually dependent. This mutual dependence is controlled using the settings described in Table 6-2.

Table 6-2 Settings that control project and connection interdependence

Level of control	Setting	Description
Connection	Project	This is a setting that you specify for a connection. It identifies one or more projects that are to be opened automatically when you connect to a target processor on that connection. For more details, see the appendix that describes the connection properties in the <i>RealView Debugger v1.8 Target Configuration Guide</i>.
Project	Specific_device	This is a setting that you specify for a project. It identifies a specific processor (for example, ARM7TDMI®) or processor family (for example, ARM7). This restricts the loading of the image for this project to a connection with the specified processor or processor family. RealView Debugger acts on this setting only when: • a connection to a matching target processor already exists, and you open the project • the project is already open, and you connect to a matching target processor. This close coupling of project and target processor is referred to as <i>specific device binding</i> or <i>autobinding</i>. It is particularly useful if you are working with multiple projects open. For more details, see the chapter that describes managing projects in the <i>RealView Debugger v1.8 Project Management User Guide</i>. Other project-related settings are provided that enable you to control what happens when a project interacts with a connection (see <i>Automatic operations performed by a project</i>).

6.1.5 Automatic operations performed by a project

A RealView Debugger project enables various operations to be performed automatically. To enable these operations to be performed automatically, RealView Debugger associates the project with a connection using a mechanism called *binding*. However, RealView Debugger only binds a project to a connection in the following circumstances:

- when a connection already exists, and the project you open is the first one opened in the current debugging session
- when one or more projects are open, and you connect to a target processor

- when you open an autobound project, that is a project associated with a specific device (see Table 6-2 on page 6-6), and a connection has a target processor that matches the device.

If you are working with multiple projects, there are specific rules that RealView Debugger uses to determine which project to bind to a connection. This might involve displaying a prompt to which you must respond.

If you do not restrict image loading to a specific device (see Table 6-2 on page 6-6), then the binding mechanism is called *default binding*.

In addition to invoking these automatic operations, project binding has other effects when you are working with projects. See the chapter that describes managing projects in the *RealView Debugger v1.8 Project Management User Guide* for full details on project binding.

Setting up the automatic operations for a project

You set up the operations in the Project Properties (see *Project properties* on page 6-3). Table 6-3 lists these operations and shows you where to find the information to implement them.

Table 6-3 Project-related operations

Operation	Reference ^a
Load the image associated with a project, if it exists.	See the Detailed Description of the Open_load setting
Set the initial load state of the image, which is one of: • register image name only • load symbols and image • load symbols only.	See the Detailed Description of the Open_load setting
Set any image-related controls. For example, set the program counter (PC) to the image entry point.	See the Detailed Description of the settings in the Image_load group
Set various runtime controls, such as top_of_memory, and command-line arguments if the image accepts these.	See the Detailed Description of the settings in the Runtime_Control group
Set one or more predefined breakpoints.	See the Detailed Description of the settings in the Auto_Set_Breaks and Named_Breaks groups
Run one or more RealView Debugger CLI commands.	See the Detailed Description of the settings in the Command_Open_Close group

a. To view the Detailed Description of a setting, right-click on the setting and select **Detailed Description...** from the context menu.

6.2 Setting up project build options for the basic compilation tasks

This section describes how to set up the build options for a project to perform the basic compilation tasks. It includes the following sections:

- *Targeting the source language* on page 6-9
- *Targeting the ARM or Thumb instruction set* on page 6-10
- *Targeting a specific ARM architecture or processor* on page 6-11
- *Targeting specific procedure call standard variants* on page 6-11
- *Generating debug information* on page 6-12
- *Optimizing your compiled sources* on page 6-12
- *Building an image with a single compiler invocation* on page 6-13
- *Building an image with separate command invocations* on page 6-13
- *Specifying the initial entry point for an image* on page 6-13
- *Creating object libraries* on page 6-13
- *Converting images to binary files* on page 6-14
- *Creating an image memory map with scatter-loading* on page 6-14
- *Project settings for the example Dhrystone project* on page 6-14.

The compilation tools are described in detail in the following documents:

- *RealView Compilation Tools Assembler Guide*
- *RealView Compilation Tools Compiler and Libraries Guide*
- *RealView Compilation Tools Linker and Utilities Guide.*

———— Note ————

If you create a RealView Debugger project, you can set up the build tool options using the RealView Debugger GUI. See the chapter that describes managing projects in the *RealView Debugger v1.8 User Guide*. Also, see *Help on creating RealView Debugger projects* on page 6-2, which provides information to help you decide the type of RealView Debugger project to create.

6.2.1 References to project settings

The following sections reference the settings in the Project Properties using the following syntax:

`[group.[group.]]setting`

The period indicates that the following item is a sub-option of the preceding item. For example, in the `COMPILE=arm` settings, `Compilation.Checking.Source_language` refers to the `Source_language` setting in the `Checking` subgroup of the `Compilation` group.

6.2.2 Targeting the source language

To target a specific source language, use the project settings shown in Table 6-4.

Table 6-4 Settings for targeting the source language

Language	COMPILE Settings	Value
ISO standard C	Compilation.Compiler	ARM_C
	Compilation.Checking.Source_language	C90 ^a
Strict ISO standard C	Compilation.Compiler	ARM_C
	Compilation.Checking.Source_language	C90
	Compilation.Checking.Strict_iso	strict
ISO standard C++	Compilation.Compiler	ARM_Cpp
	Compilation.Checking.Source_language	CPlusPlus
Strict ISO standard C++	Compilation.Compiler	ARM_Cpp
	Compilation.Checking.Source_language	CPlusPlus
	Compilation.Checking.Strict_iso	strict

a. This is the default value.

When you add source files to a project, RealView Debugger places the sources in the correct COMPILE group:

- The COMPILE=arm group contains C sources.
- The COMPILE=arm_cpp group contains C++ sources.

In both cases, the correct Compilation.Compiler setting is set by default. However, you must set the strict ISO setting manually, if this is required.

Also, RealView Debugger places any assembly language sources you add in the ASSEMBLE=arm group.

6.2.3 Targeting the ARM or Thumb instruction set

You can compile or assemble your sources for the ARM or Thumb instruction set. Table 6-5 shows the settings that enable you to do this.

Table 6-5 Settings for targeting ARM and Thumb

Target instruction set	COMPILE and ASSEMBLE Settings	Value
ARM C	Compilation.Compiler Compilation.Checking.Source_language	ARM_C C90 ^a
ARM C++	Compilation.Compiler Compilation.Checking.Source_language	ARM_Cpp CPlusPlus
ARM assembly	Assembly.Assemble_mode	ARM
Thumb C	Compilation.Compiler Compilation.Checking.Source_language	Thumb_C C90
Thumb C++	Compilation.Compiler Compilation.Checking.Source_language	Thumb_Cpp CPlusPlus
Thumb assembly	Assembly.Assemble_mode	Thumb

a. This is the default value.

If any of your sources are Thumb C, Thumb C++, or Thumb assembler, RealView Debugger places these in the corresponding arm group. If these sources require the --thumb option when compiling or assembling, you must move them to the corresponding thumb group. For example, if you have a source file thumb.c, this is placed in the COMPILE=arm group, You must remove the file from this group, and add it to the COMPILE=thumb group.

However, if your sources use the per-function instruction (#pragma thumb) and directive (CODE16), you do not have to move them, because these override the settings during the build.

If you include the pragmas or directives in your source code, you generate mixed instruction set code. For this to work properly, you must also use the interworking compiler option. See *Targeting specific procedure call standard variants* on page 6-11 for details.

To see how the assembler directives are used, examine the RVDS example assembler file:

```
install_directory\RVDS\Examples\...\asm\thumbsub.s.
```

6.2.4 Targeting a specific ARM architecture or processor

To compile or assemble your sources for a specific ARM architecture or processor, use the settings of the `COMPILE=` and `ASSEMBLY=` groups shown in Table 6-6.

Table 6-6 Targeting a specific ARM architecture or processor

Target type	COMPILE and ASSEMBLE Settings	Example value
ARM architecture	Compilation.Optimization.Cpu Assembly.Cpu	4T
Processor	Compilation.Optimization.Cpu Assembly.Cpu	ARM7TDMI

For more details about the `--cpu` option, see the *RealView Compilation Tools Assembler Guide* and the *RealView Compilation Tools Compiler and Libraries Guide*.

6.2.5 Targeting specific procedure call standard variants

You can specify one or more procedure call standard variants.

Interworking

If your application is built from sources that target both the ARM and Thumb instruction sets, you must specify interworking:

`Compilation.APCS.Interworking enabled`

`Assembly.APCS.Interworking enabled`

To see how these options are used, examine the RVDS example project:

`install_directory\RVDS\Examples\...\interwork`

Read-only position independence

To specify read-only position independence when compiling or assembling, use the option:

`Compilation.APCS.Ropi enabled`

`Assembly.APCS.Ropi enabled`

You can specify read-only position independence when linking with the options:

`Link_Advanced.Ropi enabled`

`Link_Advanced.Ro_base address`

Read/write position independence

To specify read/write position independence when compiling or assembling, use the option:

`Compilation.APCS.Rwpi enabled`

`Assembly.APCS.Rwpi enabled`

You can specify read/write position independence when linking with the options:

`Link_Advanced.Rwpi enabled`

`Link_Advanced.Rw_base address`

For more details about these, and other `--apcs` options, see the *RealView Compilation Tools Assembler Guide*, the *RealView Compilation Tools Compiler and Libraries Guide*, and the *RealView Compilation Tools Linker and Utilities Guide*.

6.2.6 Generating debug information

If you want to debug your image using RealView Debugger, specify debug option:

`Compilation.Generate_debug enabled`

`Assembly.Generate_debug enabled`

The compiler also provides a pragma equivalent, `#pragma debug` (see the *RealView Compilation Tools Compiler and Libraries Guide* for details).

By default, the linker includes debug information (see the description of the `-debug` option in the *RealView Compilation Tools Linker and Utilities Guide*).

6.2.7 Optimizing your compiled sources

When you are compiling C or C++ code, the compiler enables you to optimize the generated code. Some optimizations are performed by default. However, you can control which optimizations you want to apply to your code.

The compiler optimizations are available in the Optimization settings group. For example, setting `Optimization.Debug_optimize` to maximum gives maximum optimization.

The compiler also provides pragma equivalents for these optimizations.

For more details on the optimizations that are available, see the *RealView Compilation Tools Compiler and Libraries Guide*.

6.2.8 Building an image with a single compiler invocation

Standard and Library project are unable to generate images with a single compiler invocation. To do this, you must create a makefile containing the required compiler command, and then create a Custom project to run this makefile. See *Limitations of Standard and Library projects* on page 6-5 for more details.

By default, the compiler attempts to generate an image from the specified source files. The default image name is `__image.axf`. To specify your own image name, use the `-o image_name` option. For example:

```
armcc f1.c f2.c -o myimage.axf
```

For more details, see the *RealView Compilation Tools Compiler and Libraries Guide*.

6.2.9 Building an image with separate command invocations

For Standard and Library projects, RealView Debugger always generates makefiles that use separate command invocations to compile, assemble, and link. In addition, it uses a separate compile or assemble command for each source file you specify. See *Limitations of Standard and Library projects* on page 6-5 for more details.

6.2.10 Specifying the initial entry point for an image

If your image contains multiple entry points, you must specify a unique initial entry point for the image. To do this, use the following BUILD setting:

```
Link_Advanced.Entry entry_point
```

For example:

```
Link_Advanced.Entry=0x8000
```

For more details, see the *RealView Compilation Tools Linker and Utilities Guide*.

6.2.11 Creating object libraries

The ARM librarian utility provided with RealView Compilation Tools enables you to create object libraries. For example, to create a library called `mylib` containing all object files in the current directory, you might use the command:

```
armar -create mylib *.o
```

You can create an ARM library in one of the following ways:

- For a Standard project, run the ARM librarian as a Post link command. To do this, use the `Pre_Post_Link.Post_link` setting in the BUILD group. For example:

```
Pre_Post_Link.Post_link "armar -create mylib *.o"
```

- Create a Library project. This type of project creates object files from your sources, then runs the armar utility to create the library. You can configure the options to the armar utility in the BUILD_LIB settings group.

For more information on using the ARM librarian, see the *RealView Compilation Tools Linker and Utilities Guide*.

6.2.12 Converting images to binary files

The fromelf utility provided with RealView Compilation Tools enables you to convert your images to different formats. For example, to convert the image infile.axf to a plain binary file outfile.bin for downloading to flash, you might use the command:

```
fromelf -bin -o outfile.bin infile.axf
```

You can set up this command to run after the link stage. To do this, use the Pre_Post_Link.Post_link setting in the BUILD group. For example:

```
Pre_Post_Link.Post_link "fromelf -bin -o outfile.bin infile.axf"
```

For more information on using the fromelf utility, see the *RealView Compilation Tools Linker and Utilities Guide*.

6.2.13 Creating an image memory map with scatter-loading

To create an image memory map, you link using a scatter-load description file. To specify a scatter-loading file use the following BUILD setting:

```
Link_Advanced.Sscatter_file scatter_filename
```

To see how scatter-loading is used, examine the RVDS example project:

```
install_directory\RVDS\Examples\...\emb_sw_dev
```

For more details on creating and using scatter-load description files, see the *RealView Compilation Tools Developer Guide* and the *RealView Compilation Tools Linker and Utilities Guide*.

6.2.14 Project settings for the example Dhrystone project

The following commands and options are used to build the example Dhrystone project:

```
armcc -c -DMSC_CLOCK -W -g --dwarf2 -Otime -O1 dhry_1.c -o Debug\dhry_1.o
armcc -c -DMSC_CLOCK -W -g --dwarf2 -Otime -O1 dhry_2.c -o Debug\dhry_2.o
armlink Debug\dhry_1.o Debug\dhry_2.o -o Debug\dhrystone.axf --info totals
```

Note

Because both source files are written in C, the compiler automatically compiles using ISO standard C. Therefore, the `--c90` compiler option is omitted.

Table 6-7 describes the project build options to use in RealView Debugger. RealView Debugger always includes the `-c` compiler switch in the generated makefiles, so there is no equivalent project build option. This instructs the compiler to only generate object files.

Table 6-7 Build options used to build the dhrystone example project

Project build option	Value	Description
Messages.Suppress_all	enabled	Instructs the compiler to suppress all warning messages. This inserts the <code>-W</code> compiler switch.
Compilation.Generate_debug	enabled	Instructs the compiler to include debug tables. This inserts the <code>-g</code> compiler switch.
Optimization.Debug_optimize	partial	Instructs the compiler to generate partially optimized code. This inserts the <code>-O1</code> compiler switch.
Optimization.Speed_vs_space	speed	Instructs the compiler to perform optimizations to reduce execution time at the possible expense of a larger image. This inserts the <code>-Otime</code> compiler switch.
Compilation.Debug_format	dwarf2	Instructs the compiler to create DWARF2 debug information table format. This inserts the <code>--dwarf2</code> compiler switch. This is the default and only option.
Preprocessor.Define	MSC_CLOCK	Instructs the compiler to define the symbol <code>MSC_CLOCK</code> as a preprocessor macro. This inserts the <code>-D</code> compiler switch.
BUILD.Application	dhrystone.axf	Instructs the linker to create an image with the filename <code>dhrystone.axf</code> . This inserts the <code>-o</code> linker switch.
BUILD.Messages.Sizes	totals	Instructs the linker to display the totals of the Code and Data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries. This inserts the <code>--info totals</code> linker switch.

See the *RealView Compilation Tools Compiler and Libraries Guide* for a description of the compiler options.

See the *RealView Compilation Tools Linker and Utilities Guide* for a description of the linker options.

Chapter 7

Getting Started with the RealView Debugger Command Line Interface

This chapter describes how to get started with the RealView® Debugger CLI. It contains the following sections:

- *About the CLI* on page 7-2.
- *Entering commands interactively in the GUI* on page 7-5
- *Entering batched commands* on page 7-6
- *Using the headless debugger* on page 7-10.

7.1 About the CLI

This section introduces the CLI and contains the following sections:

- *The CLI interfaces of RealView Debugger*
- *Why use the CLI?* on page 7-3
- *Accessing the CLI from the GUI* on page 7-3
- *Getting help* on page 7-4.

7.1.1 The CLI interfaces of RealView Debugger

RealView Debugger provides two interfaces that enable you to perform all supported operations:

GUI The *Graphical User Interface* (GUI) contains a main Code window, where you can access the majority of features. There are also other dialogs and panes you can access from the Code window. For complete details on these elements of the RealView Debugger GUI, see Chapter 4 *RealView Debugger Desktop*.

CLI Enables you to enter commands directly into the debugger without requiring the use of GUI items. This chapter provides an introduction to the components of the command language that you must read if you want to perform debugger operations from the command line. You can enter commands:

- in the **Cmd** tab of the Output pane when using the GUI (see *Accessing the CLI from the GUI* on page 7-3)
- at the command prompt when using the headless debugger (see *Using the headless debugger* on page 7-10).

See the chapter that describes the RealView Debugger commands in the *RealView Debugger v1.8 Command Line Reference Guide* for the full list of commands you can enter.

7.1.2 Why use the CLI?

Although you can perform the complete set of RealView Debugger operations from the GUI, there are specific advantages to using the CLI depending on your requirements, such as:

Setting up the system state

You can use scripted CLI commands, or include files, to place a target in a particular state, either whenever you connect to a target or on demand. For example, you might use a script to initialize peripherals, or to zero video memory.

Running test scripts

Test scripts are used to check that programs are behaving according to specification. You can use memory reads, writes, and target function calls in the CLI to exercise a module API. You can log semihosting output and check that against known correct output, and you can use the debugger to perform soak tests, using many I/O cycles to check system stability.

Performing test script conversion

The CLI enables you to enter commands that are equivalent to `armsd` and `AXD` commands. This is particularly useful when you want to convert test scripts you were running with the ADS. For details on converting commands, see:

- *Converting legacy AXD scripts to RealView Debugger format* on page B-16
- *Converting legacy armsd scripts to RealView Debugger format* on page C-5.

7.1.3 Accessing the CLI from the GUI

The CLI command line in the GUI is available in the **Cmd** tab of the Output pane (see Figure 4-1 on page 4-3).

Note

Many operations you perform with the RealView Debugger GUI, such as loading an image, generate equivalent CLI commands in the **Cmd** tab. This is a useful aid to understanding how to use the CLI commands. In addition, if you have turned on logging, these commands are also written to the log file. You can then use the log file as a command script (see *Capturing commands and output* on page 7-14 for more information).

7.1.4 Getting help

When you are running the RealView Debugger, you can display details of commands using either of the following:

- *Help on Commands*
- *DCOMMANDS command*.

Help on Commands

To display help on the commands:

1. Select **Help on Commands** from the **Help** menu in the Code window. This displays the Commands topic, that lists all the available commands.
2. Click on the required command to see a brief description of that command.

DCOMMANDS command

The DCOMMANDS command, and its alias DHELP and short form DCOM displays the syntax or full description of a single command, a group of related commands, or all available commands. There are some commands that DCOMMANDS does not have any information about. For information about these commands see the chapter that describes the RealView Debugger commands in the *RealView Debugger v1.8 Command Line Reference Guide* or use the online help option **Help on Commands**.

7.2 Entering commands interactively in the GUI

You can enter commands directly onto the GUI command line (see Figure 4-1 on page 4-3).

Note

You can also enter commands using the headless debugger. See *Using the headless debugger* on page 7-10 for more details.

To enter a command at the command line of the GUI:

1. Make sure the Output pane is visible. To do this, select **View** → **Output** from the Code window main menu.
2. Select the **Cmd** tab in the Output pane so that the command line is given the focus. A blinking cursor appears at the command-line prompt.
3. At the prompt **>**, enter your command using either of the following methods:

Entering a new command

Type the command you want, followed by any necessary qualifiers or parameters.

Reusing a previously entered command

If you have already entered a command earlier in the session that is similar, or identical, to the command you want, you can use the up and down arrow keys to display commands previously entered. You can also select any command that RealView Debugger has inserted automatically as a result of other GUI operations.

To edit your command, you can use the left and right arrow keys to position the cursor at the desired location in the command, and the Backspace and Delete keys to delete characters.

Note

The total length of a command must not exceed 4095 characters.

4. When you have entered the desired command, press Enter to execute the command.

7.3 Entering batched commands

This section describes the various ways that you can include, and execute, batch (script) files while working with RealView Debugger. You can also use the CLI to invoke a batch file.

You can use RealView Debugger to execute commands from batch files in the following ways:

- *Using the -inc command-line option*
- *Using the INCLUDE command on page 7-7*
- *Using the Command setting in the board file advanced information block on page 7-7*
- *Adding comments to your batch files on page 7-8*
- *Using the QUIT command in batch files on page 7-9.*

7.3.1 Using the -inc command-line option

RealView Debugger supports its own command-line options, that are accessible using the Target text field of the Windows shortcut Properties dialog, or from the Sun Solaris, Red Hat Linux, or Windows shell. These CLI options include -inc, that identifies the batch file to run on startup, and -b, that runs the debugger in batch mode.

Note

You must specify the install location using the -install option if the debugger is not in the default directory specified in the environment variable RVDEBUG_INSTALL.

To use -inc:

1. Construct a script file for your session:
 - Use a text editor to create the script file.
 - Use the JOURNAL command in a debugging session to log commands to a file.
2. If the debugger is running, then exit the debugger.
3. Use a Windows shortcut, or a command shell, to run the debugger in one of the following ways:
 - with the -b option, to execute the commands in the file in batch mode and then exit the debugger, for example:
`rvdebug -install="installdir" -b -inc="\path\to\file.txt"`

Note

Do not use -b without -inc.

- with the `-cmd` option, to execute the commands in the file and then leave the debugger running ready to continue the debugging session in command-line mode
- without the `-b` and `-cmd` options, to execute the commands in the file and then leave the debugger running ready to continue the debugging session in GUI mode.

The CLI startup options are explained in the chapter that describes starting to use RealView Debugger in *RealView Debugger v1.7 User Guide*.

7.3.2 Using the INCLUDE command

The INCLUDE command executes commands that are stored in an include file you specify.

Note

If you want to include a batch file when a target is running, you must first enter the `wait=off` command, then include the batch file:

```
> wait=off
> include myfile.inc
```

Your batch file can still include the `wait=on` command, if required.

7.3.3 Using the Command setting in the board file advanced information block

You can configure a board file for each target processor you use with RealView Debugger. The *Extended Target Visibility* (ETV) definitions for each target are held in the Advanced_Information block (see *RealView Debugger v1.7 Target Configuration Guide* for more information).

In the advanced information block, you can use the Command setting to run a single RealView Debugger command when a connection is established. You can specify the INCLUDE command in this setting to run more commands from a file. For example, you might want to load an image to the connected target, and specify breakpoints for debugging.

Note

Specifying an INCLUDE command in the board file settings requires you to amend your `.brd` file, usually called `rvdebug.brd`, located in your default home directory. It is strongly recommended that you make a backup of this file before editing your configuration so that you can restore your configuration file.

7.3.4 Adding comments to your batch files

You can add comments to your script files:

- using C or C++ style comment declarations, for example:

```
// This is a comment
/* This is another comment */
script commands
```

- starting with a semicolon, except for comments inside a macro definition, for example:

```
; command line comment
define /R void myMacro()
{
    // comment inside a macro
    macro code;
}
.
```

For more details on creating macros, see the *RealView Debugger v1.8 Command Line Reference Guide*.

- starting with `// #`, for example:

```
// # This is a comment
```

Rules for using comments in scripts

The following rules apply to comments in scripts:

- Outside a macro definition all comments must begin and end on the same line, for example:

```
; comment
// comment
/* comment */
// # comment
```

You cannot have:

```
/*
    This is a comment
*/
```

- Inside a macro definition:

- comments cannot start with `;`
- only `//` or `// #` comments can be placed at the end of a macro statement, for example:
macro_statement; // comment

- you can use `/* */` for comments, but the comment text must be a single line, for example, you can have:

```
/* This is a comment */
/*
    This is another comment
*/
```

but you cannot have:

```
/*
    This is a
    multiline comment
*/
```

Comment handling with log and journal files

If you have turned on logging or journaling, then any comments in your script files are handled as follows:

- `//` and `;` comments are not written to any log or journal file
- `// #` comments are written to journal files only, and `// #` is replaced by `;` in the journal file, for example, `// # comment` appears in the journal as:
`;` comment
- `/* */` comments are written to both log and journal files, and appear as:
 - `/* comment */` in log files
 - `> /* comment */` in journal files.

7.3.5 Using the QUIT command in batch files

If you want to use the QUIT command in a batch file, then include the `Y` parameter, for example:

```
script commands
...
quit y
```

7.4 Using the headless debugger

This section describes the commands to get you started using the headless debugger, and contains the following sections:

- *Starting the headless debugger*
- *Connecting to a debug target* on page 7-11
- *Disconnecting the current connection* on page 7-11
- *Loading an image* on page 7-12
- *Loading a Flash image* on page 7-12
- *Displaying the current working directory* on page 7-14
- *Capturing commands and output* on page 7-14
- *Working with multiple connections* on page 7-15.

See the chapter that describes the RealView Debugger commands in the *RealView Debugger v1.8 Command Line Reference Guide* for the full list of commands you can enter.

7.4.1 Starting the headless debugger

To start the headless debugger, do the following:

- On Windows, enter the following command, either from the **Start** → **Run** dialog or a DOS command window:

```
rvdebug -cmd
```

A DOS command window is displayed.

- On Sun Solaris or Red Hat Linux, enter the following command at the shell prompt:

```
rvdebug -cmd
```

In both cases, the following prompts are displayed:

```
...
Enter commands, or 'dhelp=all' for list, or 'quit' to exit
TVS>
```

When using the headless debugger, you do not have access to the GUI.

Starting the debugger with a connection and an image loaded

To start the headless debugger with a connection and an image loaded, use the `-init` and `-exec` command options. For example, to connect to RVISS and load the image `C:\testimages\testimage.axf`, enter:

```
rvdebug -cmd -init=@new_arm@localhost -exec="C:\testimages\testimage.axf"
```

For more details on using these options, see *RealView Debugger v1.8 User Guide*.

7.4.2 Connecting to a debug target

Use the **CONNECT** command to connect to a debug target. You must specify the *access-provider connection*, then the *endpoint connection*.

————— Note —————

Use named connections when using the headless debugger.

For example, to connect to *RealView ARMulator® ISS* (RVISS) through the localhost Simulator Broker:

1. Connect to the route (access-provider connection):

```
connect,route @localhost
```

The following messages are displayed:

```
= 3. Board Entries added/modified
TVS>
```
2. Connect to RVISS (endpoint connection):

```
TVS> connect @new_arm
```

```
= 4. CONNECT: Board Simarm_1 (Sim) is now connected
...
Stop>
```

7.4.3 Disconnecting the current connection

Use the **DISCONNECT** command to disconnect from the current connection.

For example, to disconnect from the RVISS connection enter:

```
disconnect
```

If this is your only connection, the following messages are displayed:

```
= 36. CONNECT: Board Simarm_1 (Sim) is now disconnected
Stop> = 37. CURRENT: No Current Board
TVS> = 38. Board Entries added/modified
TVS>
```

If you have more than one connection, then the next connection becomes the current connection. This is identified in the displayed messages, for example:

```
= 36. CONNECT: Board Simarm_2 (Sim) is now disconnected
Stop> = 37. CURRENT: New Current Board is Simarm_1:Sim (P1) {@Simarm_1:Sim}
TVS> = 38. Board Entries added/modified
TVS>
```

See *Working with multiple connections* on page 7-15 for more details.

7.4.4 Loading an image

Use the LOAD command to load an image to the current target. For example, to load the example dhrystone image from the default installation directory on a Windows system, enter:

```
load 'C:\Program Files\ARM\RVDS\Examples\2.2\10\windows\dhrystone\Debug\dhrystone.axf'
```

The following messages are displayed:

```
Loading file C:\Program Files\ARM\RVDS\Examples\2.2\10\windows\dhrystone\Debug\dhrystone.axf...
Stop>
```

7.4.5 Loading a Flash image

To load an image into Flash:

1. Connect to your target hardware. For example, to connect to an ARM940T with RealView ICE, enter the command:

```
connect @ARM940T_0@RealView-ICE
```
2. Specify the Flash memory area for your target device, and the Flash Method (FME) file to use (see *Specifying the Flash memory area*).
If you are using one of supported devices (for example, Integrator/AP), use the FME file for the device that is provided with RVDS. Otherwise, you must create your own FME file. For instructions on how to create an FME file, see the chapter that describes programming Flash with RealView Debugger in the *RealView Debugger v1.8 Target Configuration Guide*.
3. Load your image into Flash (see *Loading the image into Flash* on page 7-13).
4. View Flash information (see *Displaying Flash information* on page 7-14).

Specifying the Flash memory area

Before you can load an image into Flash, you must specify:

- the Flash memory area for your target device
- the FME file to use.

To do this, use the MEMMAP command.

For example, if you have an Integrator/AP board it has an Intel Flash memory, uses 4-byte memory accesses, and starts at address 0x24000000 with length 0x2000000. The FME file, flash_IntegratorAP.fme, is located in the Flash examples directory:

```
install_directory\RVD\Core\...\flash\examples\IntegratorAP
```

Therefore, to set the Flash memory for the target, enter the command:

```
mmap,def,access:Flash,type:Any,asize:4,descr:"Intel",fme:"install_directory\RVD\
Core\...\flash\examples\IntegratorAP\flash_IntegratorAP.fme"=0x24000000..+0x2000
000
```

In this example, replace *install_directory* and ... with the values for your installation.

Loading the image into Flash

Use the READFILE command to load a Flash image to the current target. For example, to load the Flash image for a project in C:\My Projects\flash\Debug on a Windows system at the Flash address 0x24000000, enter:

```
readfile,OBJ "C:\My Projects\flash\Debug\myflash.axf"=0x24000000
```

Messages similar to the following are displayed:

```
= 17. CURRENT:
Conn> = 18. PCREATE: Monitor process 1 auto-attached
Stop> = 19. CURRENT: Current Process on board ARM940T_0:A is 1: <no name>
Stop> Flash block 7 opened for modify.
Stop> Flash block 6 opened for modify.
Stop> Flash block 5 opened for modify.
Stop> Flash block 4 opened for modify.
Stop> Flash block 3 opened for modify.
Stop> Flash block 2 opened for modify.
Stop> Flash block 1 opened for modify.
Stop> Flash block 0 opened for modify.
Stop> Flash block 10 opened for modify.
Stop> Flash block 9 opened for modify.
Stop> Flash block 8 opened for modify.
Stop> = 20. UPDATE: Memory modified from file read.
Stop> Flash opened on ARM940T_0:ARM-ARM-NW for 'Intel DT28F320S3 2Mx16 x2 x4' at
'0x24000000'
C:\My Projects\flash\Debug\myflash.axf
Stop>
```

Displaying Flash information

To display Flash information, enter the FLASH command without any qualifiers or parameters.

7.4.6 Displaying the current working directory

RealView Debugger maintains a current working directory:

- When you load an image directly to create an auto-project, the current working directory changes to the location of the image.
- If you are using the RealView Debugger GUI, and you open a project, the current working directory changes to the location of the project (.prj) file.

The current working directory is always the location of the last image loaded as an auto-project, or the last project opened.

To find the current working directory use the HOST command, together with the appropriate command for your operating system. For example, on Windows enter:

```
host cd
C:\myproject\Debug
```

———— Note ————

You cannot use the HOST command to change the current working directory.

7.4.7 Capturing commands and output

You can capture the commands you enter at the CLI, and any messages generated by RealView Debugger to a text file. RealView Debugger provides the following commands to enable you to do this:

- LOG captures the commands you enter. You can then use the log file as an include file.
- JOURNAL captures the commands you enter, and any output that is generated by the commands or macros, including error messages.
- STDIOLOG captures messages and prompts that your application sends to STDIO.

———— Note ————

Some commands and macros accept window or file IDs. Window IDs identify windows that you have opened with the VOPEN command, and so are only available when you are using the RealView Debugger GUI.

If you have a journal or log file open, you can specify the appropriate ID, so that text is sent directly to the file. You can also direct output to STDIO. If you are using the RealView Debugger GUI, this is displayed in the **StdIO** tab of the Output pane, and any open STDIO log file.

7.4.8 Working with multiple connections

If you have a multiprocessor debugging license, you can establish multiple connections. Each new connection you make to a debug target becomes the *current connection*. This is the connection that is used when you enter connection-related commands, for example, when you load an image.

The following tasks might help you when you are working with multiple connections:

- *Displaying information about the current connection*
- *Identifying other connections*
- *Disconnecting a named connection* on page 7-16
- *Changing the current connection* on page 7-16.

Displaying information about the current connection

Use the DTBOARD command to display information about the current connection, for example:

```
dtboard
Connected Board 'Simarm_1' Port 0: Server supporting Single Tasking.
  Port string: localhost
  Entry of router/broker localhost
```

Identifying other connections

To find out the names of your connections use the BOARD command with the ,next parameter to determine the next connection, for example:

```
board,next
= 39. CURRENT: New Current Board is Simarm_2:Sim (P1) {@Simarm_2:Sim}
Stop> New Current Board is Simarm_2:Sim (P1)
```

Note

This command also changes the current connection. Therefore, cycle through the connections until the details of your original current connection is displayed. See *Changing the current connection* on page 7-16 for more details.

Disconnecting a named connection

You might want to disconnect a connection that is not the current connection. To do this, you must specify the name of the connection, for example:

```
disconnect @Simarm_2
= 40. CONNECT: Board Simarm_2 (Sim) is now disconnected {@Simarm_2:Sim}
Stop> = 41. Board Entries added/modified
Stop>
```

If you have more than one connection, then the next connection becomes the current connection. This is identified in the displayed messages, for example:

```
= 42. CONNECT: Board Simarm_2 (Sim) is now disconnected {@Simarm_2:Sim}
Stop> = 43. CURRENT: New Current Board is Simarm_1:Sim (P1)
Stop> = 44. Board Entries added/modified
Stop>
```

Changing the current connection

To perform a connection-related command on a connection other than the current connection, you must make that connection current. To do this, enter:

```
board ="target_name:access-provider"
```

For example:

```
board ="Simarm_2:localhost"
= 45. CURRENT: New Current Board is Simarm_2:Sim (P1) {@Simarm_2:Sim}
Stop> New Current Board is Simarm_2:Sim (P1)
Stop>
```

Alternatively, if you are not sure of the connection name, you can cycle through the connections using the ,next qualifier, for example:

```
board,next
= 46. CURRENT: New Current Board is Simarm_2:Sim (P1) {@Simarm_2:Sim}
Stop> New Current Board is Simarm_2:Sim (P1)
```

Each time you enter this command, the next connection becomes the current connection. Repeat this command until the required connection is current.

Appendix A

Configuration Files Reference

This appendix describes the files set up for a new installation of RealView® Debugger v1.8, where they are stored, and what information each file holds. This appendix assumes that you have chosen a Typical installation. It contains the following sections:

- *Overview* on page A-2
- *Files in the etc directory* on page A-3
- *Files in the home directory* on page A-5.

A.1 Overview

RealView Debugger creates files containing default settings and target configuration information when you first install the product. The files created (or modified) depend on what kind of installation you have chosen. If you are upgrading from an earlier version of RealView Debugger, some files are modified so that previous configuration information is not lost.

When you run the debugger for the first time, some of this information is copied into the RealView Debugger default home directory. This is an empty directory created by the installation process. If you are upgrading from an earlier version of RealView Debugger, the installer creates the home directory and copies your existing configuration details into it. This means that this area is not empty when you run the debugger for the first time.

Note

This appendix describes configuration files created (or modified) when you install RealView Debugger on Windows. The *RealView Debugger v1.8 User Guide* contains information for developers using RealView Debugger on non-Windows platforms. See Appendix B *RealView Debugger for Sun Solaris and Red Hat Linux* for details.

A.2 Files in the etc directory

When you install RealView Debugger v1.8, files containing default settings and target configuration details are installed in the default settings directory \etc. What files are installed, and their contents, depend on what other software is detected during the installation, for example *RealView Compilation Tools* (RVCT), or *ARM Developer Suite™* (ADS):

armul.var	A list of RealView ARMulator® ISS variants used by RealView Debugger. You can edit this file to add hand-made variants.
genmake.loc	Specifies the location of the development tools used when working with projects. RealView Debugger copies this file into the default home directory (when you first create a project) or merges project details (when you first work with an existing project). Do not edit this file manually.
gen_arx.mk	Toolchain-specific template used by the genmake program to generate makefiles. You can edit this file to modify the input/output of the genmake program.
genmake_*.loc	Toolchain-specific makefile template used to define the location of development tools, for example genmake_ARM_ADS.loc or genmake_C2.loc. Do not edit these files manually.
pARM*.prc	Processor-specific information files used to define support for emulators and simulators, and for project information, for example pARM.prc, pARM_C2.prc, or pARM_RVI.prc. Do not edit these files manually.
rvdebug.brd	The default board file used to contain target configuration settings. This file references .bcd and .jtg files.
rvdebug.tpl	Template file used to contain standard templates. You can edit these files for use in source files.
startup.mk	MSDOS DMAKE startup file. You can edit this file if required.
targ_*.aco	Processor-specific instruction format files used by RealView Debugger to color assembler code in the File Editor, for example targ_ARM.aco. Do not edit this file manually.

- `template.spr` SPR template used by RealView Debugger.
Do not edit this file manually.
- `*.bcd` Board/Chip definition files supplied by hardware manufacturers.
If you want to make changes to these files, copy them into your default home directory and edit them using the Connection Properties window, for example `CM740T.bcd`.
- `*.jtg` JTAG configuration files, for example `arm.jtg`.
- `*.stp` RealView Debugger internal template files containing debugger defaults. These are used as internal support files or as template files for settings. For example, they are used to support project properties and define options such as workspace settings, for example `prj_set.stp`.
Do not edit these files manually.

For a new installation, some of these files are copied into your RealView Debugger default home directory when the debugger runs for the first time.

A.3 Files in the home directory

When you install and run RealView Debugger v1.8 for the first time, files containing default settings and target configuration details are copied into your RealView Debugger default home directory where they can be maintained.

If you are upgrading from an earlier version of RealView Debugger, the installer creates the home directory and copies your existing configuration details into it. This means that this area is not empty when you run the debugger for the first time:

<code>armreg.sig</code>	An internal settings file that is created or updated each time you run RealView Debugger. Do not edit this file manually.
<code>exphist.sav</code>	Your personal history file. This file keeps a record of each session and stores your personal favorites, for example watched variables. This file is updated at the end of each debugging session.
<code>genmake.loc</code>	Specifies the location of the development tools used when working with projects. RealView Debugger copies this file into the default home directory when you first create a project.
<code>rvdebug.aws</code>	Your default workspace settings file. For a new installation, RealView Debugger creates this file when it runs for the first time. By default, this file is updated with the state of the debugger at the end of each debugging session.
<code>rvdebug.brd</code>	Your board file containing target configuration settings. For a new installation, this is a duplicate of the file installed in the default settings directory \etc. This file references <code>.rbe</code> files and <code>.bcd</code> files.
<code>rvdebug.ini</code>	Specifies global configuration settings used across all workspaces or when working without a workspace. For a new installation, RealView Debugger creates this file when it runs for the first time.
<code>rvdebug.sav</code>	This file specifies how each RealView Debugger session starts. For a new installation, RealView Debugger creates this file the first time you close down after performing an operation. By default, this file is updated at the end of each debugging session.
<code>*.auc</code>	These are default target configuration settings files created when you first run RealView Debugger, for example <code>default.auc</code> . These files are referenced by the <code>.rbe</code> files.

- *.bcd Board/Chip definition files that you might have copied into your home directory, from the default settings directory \etc, to make changes.
Make copies of files so that the installation files are not changed.
- *.cnf These are default target configuration settings files that are created, by the RDI configuration utility, when you first run RealView Debugger, for example `armulator.cnf`.
These files are referenced by the `.rbe` files.
- *.rbe There is one `.rbe` configuration file for each available RDI target, for example `armulator.rbe`. This file is updated when you make changes to the RDI settings.
- *.rvc Configuration files for RV-msg connections. You change the contents of one of these files when you modify the configuration of an RV-msg connection using the RVConfig dialog box.
Do not edit these files manually.

A.3.1 Backup files

When you change a configuration file, RealView Debugger makes a backup copy of the current version to enable you to restore your previous settings, for example `rvdebug.aws.bak` and `rvdebug.brd.bak`. By default, backup files are given the `.bak` extension and are stored in the same location as the original file. You can change this behavior in your workspace.

Appendix B

Moving from AXD to RealView Debugger

This appendix is aimed at users who are moving development projects from *ARM® eXtended Debugger (AXD)* to *RealView® Debugger*. It includes a series of *Frequently Asked Questions (FAQs)* taken from the ARM Support website.

It contains the following sections:

- *RealView Debugger configuration* on page B-2
- *RealView Debugger operations* on page B-6
- *Project management in RealView Debugger* on page B-8
- *Comparison of RealView Debugger and AXD commands* on page B-11
- *Converting legacy AXD scripts to RealView Debugger format* on page B-16.

B.1 RealView Debugger configuration

This section answers FAQs about configuration options in RealView Debugger.

B.1.1 Where do I find Debugger Internal variables?

Select **View** → **Registers** to open the Register pane at the default location. The Debugger Internals are available from the **CycleCount** and **Semihost** tabs for the RealView Broker connections to RealView ARMulator® ISS, and the **Debug** tab for all other connections of this pane. To see a list of the available tabs, right click on the double arrows to the left of the Register pane tabs.

B.1.2 How do I set top_of_memory?

In RealView Debugger, top_of_memory is used to set the application stack base for a semihosted application running on a remote target.

If top_of_memory is not set, RealView Debugger sets it to a default value of 0x80000. A warning is displayed in the **Cmd** and **Log** tabs:

No stack/heap or top of memory defined - using defaults.

The value of top_of_memory can be overridden for a single debug session from the **Debug** or **CycleCount** tab in the Register pane (see *Where do I find Debugger Internal variables?*).

The value of top_of_memory can also be set for a particular target connection using the Connection Properties window:

1. Ensure that you disconnect from the target before making these changes.
2. Select **Target** → **Connect to Target...** to open the Connection Control window.
3. Right-click on the connection, for example RealView-ICE or Multi-ICE, and select **Connection Properties...** from the context menu.

The appropriate branch in the Connection Properties window opens automatically.

4. Expand the following entries in turn:
 - CONNECTION= (for the connection you are configuring)
 - Advanced_Information
 - Default
5. Click ARM_config in the left pane.

6. Right-click on the Top_memory entry in the right pane, select **Edit Value...** from the context menu.
7. Specify the top of memory address, for example 0x2000000, and press Enter.
8. Select **File** → **Save and Close**.

Note

To use the new setting, you must now connect to the target.

If a Board/Chip definition file is selected for this connection, then this file might contain a value for top_of_memory that overrides the target connection setting.

B.1.3 How do I configure vector_catch?

vector_catch is a mechanism used to trap processor exceptions. This feature is typically used in the early stages of development to trap processor exceptions before the appropriate handlers are installed. You select the vectors to trap by editing the vector_catch value.

The value of vector_catch can be overridden for a single debug session from the **Debug** or **CycleCount** tab of the Register pane (see *Where do I find Debugger Internal variables?* on page B-2). The value of vector_catch represents a bit field, where a set bit corresponds to a trapped exception - the LSbit corresponds to the reset vector. Although the value can only be displayed in hexadecimal, values can be entered in binary format using the notation 0b. The default value of 0x13B (0b100111011) corresponds to trapping:

Table B-1 Trapped Processor Exceptions defaults

Exception	Trapped	Comment
Reset	Yes	
Undefined	Yes	
SWI	No	SWI vector can also be trapped by the debugger to enable standard semihosting
Prefetch Abort	Yes	
Data Abort	Yes	
Reserved (Address)	Yes	This vector is not used on current ARM cores, so you do not have to trap this.

Table B-1 Trapped Processor Exceptions defaults (continued)

Exception	Trapped	Comment
IRQ	No	
FIQ	No	
Error	Yes	This a deprecated feature and performs no function

The state of individual vectors for catching can also be set for a particular target connection using the Connection Properties window:

1. Ensure that you disconnect from the target before making these changes.
2. Select **Target** → **Connect to Target...** to open the Connection Control window.
3. Right-click on the connection, for example Multi-ICE, and select **Connection Properties...** from the context menu.
The appropriate branch in the Connection Properties window opens automatically.
4. Expand the following entries in turn:
 - CONNECTION= (for the connection you are configuring)
 - Advanced_Information
 - Default
 - ARM_config
5. Click Vectors in the left pane.
6. Set the catching of each vector in the group for the connection.
7. Select **File** → **Save and Close**.

———— **Note** ————

To use the new setting, you must now connect to the target.

If a Board/Chip definition file is selected for this connection, then this file might contain a value for `vector_catch` that overrides the target connection setting.

B.1.4 How do I configure semihosting?

Semihosting is a mechanism that captures I/O requests made by code running on the target (typically library code), and communicates these to the host system for handling. For example, application printf's appear, by default, within the debugger console window. Semihosting can be enabled or disabled from the debugger.

Semihosting can be set for a single debug session from the **Debug** or **Semihost** tab of the Register pane, depending on your target (see *Where do I find Debugger Internal variables?* on page B-2). The available options, if you are using Multi-ICE®, are NO, STD (Standard), and DCC.

Semihosting can also be set for a particular target connection using the Connection Properties window:

1. Ensure that you disconnect from the target before making these changes.
2. Select **Target** → **Connect to Target...** to open the Connection Control window.
3. Right-click on the connection, for example Multi-ICE, and select **Connection Properties...** from the context menu.

The appropriate branch in the Connection Properties window opens automatically.

4. Expand the following entries in turn:
 - CONNECTION= (for the connection you are configuring)
 - Advanced_Information
 - Default
 - ARM_config
5. Click Semihosting in the left pane.
6. Set the Enabled option for the connection.
7. Select **File** → **Save and Close**.

Note

To use the new setting, you must now connect to the target.

If a Board/Chip definition file is selected for this connection, then this file might contain a value for Semihosting that overrides the target connection setting.

B.2 RealView Debugger operations

This section answers FAQs about RealView Debugger during debugging.

B.2.1 Why am I unable to connect to more than one target?

RealView Debugger is sold as a basic core product to which a number of license managed additional features can be added. Examples of additional features include multi-core debug, RTOS awareness, and support for various DSPs.

The basic version of RealView Debugger only supports single core debugging. In single core debug mode you can only connect to one target at a time. If you have to perform multi-core debugging you must purchase an additional license key to enable this feature.

B.2.2 Viewing Coprocessor variables

The RealView Debugger Register pane view includes an additional tab giving you visibility of coprocessor registers such as CP15. In RealView Debugger v1.6 the contents of these registers were displayed as hexadecimal values and not enumerated. Consult the appropriate *Technical Reference Manual* (TRM) for your ARM core to obtain information on the function of bit fields within each CP15 register.

B.2.3 How do I load symbols for an image?

To only load the symbols for an image in RealView Debugger:

1. Select **Target** → **Load Image...** to display the Load File to Target dialog box.
2. Ensure that the Symbols Only check box is selected.

B.2.4 RealView ARMulator ISS Benchmarking

Currently RealView Debugger does not support RealView ARMulator ISS benchmarking, as described in *ARM Application Note 93*.

B.2.5 Debug illusion issues

There are a number of known debug illusion issues that we are working towards fixing in future releases of RealView Debugger:

C and C++ problems

Inlined Functions:

- Stepping inlined functions might step to locations that are unexpected.
In particular this applies for STEP OVER and on the boundaries of the inlined function.

C++ problems

Exceptions:

- No support for stepping exceptions when on throw statement.
- No current support for catching exceptions (except by setting breakpoints in catch handler).

Namespaces:

- When referencing a symbol in a namespace you must qualify the symbol with the namespace, even if the PC currently points to an address in that namespace. For the example, to print the value of `mysymbol` in namespace `ns`, enter the command `printvalue ns::mysymbol`.

B.3 Project management in RealView Debugger

This section answers FAQs about project management features in RealView Debugger.

B.3.1 Can I convert CodeWarrior projects into RealView Debugger projects?

Currently it is not possible to convert CodeWarrior .mcp files directly to RealView Debugger .prj files. However, the mcp2make utility, available from the *ARM Developer Suite™* (ADS) v1.2 Downloads page, can be used to assist in converting CodeWarrior projects into makefiles. Makefiles produced in this way can then be imported into RealView Debugger, as described in *Can I use existing makefiles with RealView Debugger?* on page B-9.

B.3.2 Specifying paths to header files in projects

To do this:

1. Select **Project** → **Project Properties...** to open the Project Properties window.
2. Expand the appropriate COMPILE= group.
3. Expand the appropriate Preprocessor group.
4. Right-click on the Include entry in the right pane.
5. Select **Edit as Directory Name** from the context menu.
6. Locate the required directory and click **Select**.
7. Select **File** → **Save and Close**.

This saves the project file and regenerates the makefile.

B.3.3 Adding source files to an existing project

You can add source files to an existing project as described in the following sections:

- *From the Project Properties window*
- *From the main menu* on page B-9.

From the Project Properties window

To do this:

1. Select **Project** → **Project Properties...** to open the Project Properties window.
2. Expand the appropriate COMPILE= or ASSEMBLE= group.

3. Expand the appropriate Sources group.
4. Right-click on the Files entry in the right pane.
5. Select **Edit as Filename** from the context menu.
6. Locate the required source file and click **Save**.
7. Select **File** → **Save and Close**.

This saves the project file and regenerates the makefile.

From the main menu

Alternatively, from the main menu select **Project** → **Add Files to Project...**

Note

The Project Properties window must be closed to do this.

B.3.4 How does RealView Debugger make use of makefiles?

RealView Debugger automatically generates appropriate makefiles when projects are created or updated. Makefiles are constructed using the user-defined information in the Project Properties window and the `gen_***.mk` template located in the `\etc` subdirectory of the RealView Debugger installation. Here `***` corresponds to the default processor family and toolchain.

Generated makefiles can be viewed using any standard text editor, but hand editing of generated makefiles is not recommended because RealView Debugger automatically updates any makefiles to reflect any subsequent changes made in the Project Properties window.

B.3.5 Can I use existing makefiles with RealView Debugger?

Yes, RealView Debugger enables you to create a new project by importing an existing makefile.

Projects that are created this way differ from projects that are created entirely in RealView Debugger and you must perform additional project configuration by manually editing the makefile outside RealView Debugger.

Imported makefiles must contain the following build rules:

- `clean`
- `all`

- rebuild

Example makefile

The following example is a makefile suitable for use with RealView Debugger:

```
clean:
# for Unix:
# rm -f *.o *.axf
# for Windows
    if exist *.o del *.o
    if exist *.axf del *.axf

all: project1.axf
# for Unix
# echo Build completed
# for Windows
    if exist *.axf echo Build completed

rebuild: clean all

project1.axf: data.o io.o main.o
    armlink data.o io.o main.o -o project1.axf
io.o: io.c
    armcc -c io.c
data.o: data.c
    armcc -c data.c
main.o: main.c
    armcc -c main.c
```

B.4 Comparison of RealView Debugger and AXD commands

RealView Debugger and AXD are normally driven through graphical user interfaces, but they can also be driven by typed commands. For full details of the commands available in AXD, see the *AXD and armsd Debuggers Guide*.

Some commands listed here are only similar to each other, and specific features might not be available using the listed command. If you require these features, you are recommended to check the other command descriptions for alternative ways of performing the required task.

Look up any AXD command in Table B-2 for the equivalent RealView Debugger command.

Table B-2 AXD commands and equivalents

AXD commands	RealView Debugger commands
BACKTRACE	<u>W</u> HERE
<u>B</u> REAK	<u>B</u> REAK <u>E</u> XECUTION <u>B</u> REAK <u>I</u> NSTRUCTION <u>D</u> T <u>B</u> REAK
<u>C</u> CLASSES	<u>B</u> ROWSE
<u>C</u> FUNCTIONS	<u>B</u> ROWSE
<u>C</u> LASSES	<u>B</u> ROWSE
CLEAR	<u>V</u> CLEAR
CLEARSTAT	-
CLEARWATCH	<u>C</u> LEARBREAK
<u>C</u> OMMENT <i>string</i>	<u>F</u> PRINTF
<u>C</u> ONTEXT	<u>C</u> ONTEXT <u>W</u> HERE
<u>C</u> ON <u>V</u> ARIABLES	<u>E</u> XPAND <u>S</u> COPE
<u>C</u> VARIABLES	<u>B</u> ROWSE
DBGINTERNALs	<u>E</u> DIT <u>B</u> O

Table B-2 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
<u>D</u> ISASSEMBLE	<u>D</u> ISASSEMBLE <u>M</u> ODE
ECHO	-
EXAMINE	<u>D</u> UMP
<u>F</u> ILES	-
FILLMEM	FILL <u>S</u> ETMEM
FINDSTRING	<u>S</u> EARCH
FINDVALUE	<u>S</u> EARCH
FORMAT	<u>O</u> PTION RADIX <u>S</u> ETTINGS DISASM <u>S</u> ETTINGS DSMVALUE
FUNCTIONS	<u>P</u> RINTSYMBOLS
GETFILE	<u>R</u> EADFILE
GO	<u>G</u> O
HELP	<u>D</u> COMMANDS HELP
<u>I</u> MAGES	<u>D</u> TFILE
IMGPROPERTIES	<u>D</u> TFILE
IMPORTFORMAT	-
IN	UP
LET	<u>C</u> EXPRESSION <u>S</u> ETMEM
LIST	<u>D</u> ISASSEMBLE <u>M</u> ODE <u>D</u> UMP
LISTFORMAT	-

Table B-2 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
LOAD	<u>L</u> OAD <u>R</u> ELOAD
LOADBINARY	<u>R</u> EADFILE
LOADSESSION	<u>R</u> EADBOARDFILE
LOADSYMBOLS	<u>L</u> OAD/NS
LOG	<u>J</u> OURNAL
LOWLEVEL	<u>P</u> RINTSYMBOLS
<u>M</u> EMORY	<u>D</u> UMP
OBEY	<u>I</u> NCLUDE
OUT	<u>D</u> OWN
<u>P</u> ARSE	-
PRINT	<u>P</u> RINTF <u>C</u> EXPRESSION <u>P</u> RINTVALUE
<u>P</u> ROCESSORS	<u>D</u> TPROCESS
PROCPROPERTIES	-
PUTFILE	<u>W</u> RITEFILE
<u>Q</u> UITDEBUGGER	<u>Q</u> UIT
READSYMS	<u>L</u> OAD/NS
<u>R</u> ECORD	<u>L</u> OG
REGBANKS	-
REGISTERS	-
RELOAD	<u>R</u> ELOAD
<u>R</u> UN	<u>G</u> O
RUNTOPOS	<u>G</u> O
SAVEBINARY	<u>W</u> RITEFILE

Table B-2 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
SAVESESSION	-
SETACI	-
SETBREAKPROPS	<u>B</u> REAK <u>E</u> XECUTION <u>B</u> REAK <u>I</u> NSTRUCTION
SETIMGPROP "file" cmdline	<u>A</u> R <u>G</u> U <u>M</u> E <u>N</u> T <u>S</u>
SETMEM	<u>C</u> E <u>X</u> P <u>R</u> E <u>S</u> S <u>I</u> O <u>N</u> <u>F</u> I <u>L</u> L <u>S</u> E <u>T</u> M <u>E</u> M
SETPC	<u>S</u> E <u>T</u> R <u>E</u> G @PC
SETPROC	<u>B</u> O <u>A</u> R <u>D</u> <u>T</u> H <u>R</u> E <u>A</u> D
SETPROCPROP	-
SETPROCPROP vector_catch	<u>B</u> G <u>L</u> O <u>B</u> A <u>L</u>
SETPROCPROP semihosting_enabled	
SETREG	<u>S</u> E <u>T</u> R <u>E</u> G
SETSOURCEDIR	-
SETWATCH	<u>C</u> E <u>X</u> P <u>R</u> E <u>S</u> S <u>I</u> O <u>N</u> <u>S</u> E <u>T</u> M <u>E</u> M
SETWATCHPROPS	<u>B</u> REAK <u>E</u> XECUTION <u>B</u> REAK <u>I</u> NSTRUCTION
SOURCE	<u>L</u> I <u>S</u> T
SOURCEDIR	-
STACKENTRIES	<u>W</u> H <u>E</u> R <u>E</u>
STACKIN	<u>U</u> P
STACKOUT	<u>D</u> O <u>W</u> N
<u>S</u> T <u>A</u> T <u>I</u> S <u>T</u> I <u>C</u> S	-

Table B-2 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
STEP	<u>STEP</u> INSTR <u>STEP</u> LINE <u>STEP</u> POINTSTR STEP0
STEPSIZE	RUN
STOP	STOP
TYPE	<u>L</u> IST
UNBREAK	<u>C</u> LEARBREAK
UNWATCH	<u>C</u> LEARBREAK
<u>V</u> ARIABLES	<u>P</u> rintTYPE
WATCHPT	<u>B</u> REAKACCESS <u>B</u> REAKREAD <u>B</u> REAKWRITE
WHERE	<u>C</u> ONTEXT <u>W</u> HERE

B.5 Converting legacy AXD scripts to RealView Debugger format

This section describes how you can convert legacy AXD scripts to the RealView Debugger format. The RealView Developer Suite product includes the program `axd2rvd` to help with this process.

B.5.1 `axd2rvd` command syntax

The commands have the syntax:

```
axd2rvd [-Vsn] [-A] -I infile.asd [-O rvdfile.txt]
```

where:

- | | |
|------------------------------------|---|
| <code>-I <i>infile.asd</i></code> | The name of the existing script file. If you do not specify the input file, the command prints out a usage message. There must be a space between the file option and the filename. |
| <code>-O <i>rvdfile.txt</i></code> | The name of the new RealView Debugger script file. If you do not specify this, the converted script is written to the standard output, normally the terminal. There must be a space between the file option and the filename. |
| <code>-A</code> | Include input command lines as comments in the output file. |
| <code>-Vsn</code> | Print the version number on the standard output and exit. |

You must quote filenames if they contain space characters. The filename extension used for the input or the output files is not predetermined by this program.

Note

These commands help you convert a script file to a RealView Debugger include file, but because of the different facilities provided by AXD, and the very different command set implemented, you must check the resulting file for commands that the script cannot convert.

See *Comparison of RealView Debugger and AXD commands* on page B-11 for a comparison of the commands available in AXD.

Appendix C

Moving from armsd to RealView Debugger

This appendix is aimed at users *ARM® Symbolic Debugger (armsd)* to RealView® Debugger. It contains the following sections:

- *Comparison of RealView Debugger and armsd commands* on page C-2
- *Converting legacy armsd scripts to RealView Debugger format* on page C-5.

C.1 Comparison of RealView Debugger and armsd commands

Look up any armsd command in Table C-1 for the equivalent RealView Debugger command. For full details of the commands available in armsd, see the *AXD and armsd Debuggers Guide*.

Some commands listed here are only similar to each other, and specific features might not be available using the listed command. If you require these features, you are recommended to check the other command descriptions for alternative ways of performing the required task.

Table C-1 armsd commands and equivalents

armsd commands	RealView Debugger commands
!	-
\$cmdline = "string"	<u>ARGUMENTS</u>
\$semihosting_enabled = [0 1]	BGLOBAL
\$vector_catch = [0 1]	BGLOBAL
@regname = value	<u>SETREG</u>
	# or ;
ALIAS	<u>ALIAS</u>
ARGUMENTS	<u>EXPAND</u>
BACKTRACE	<u>WHERE</u>
BREAK	<u>BREAKEXECUTION</u> <u>BREAKINSTRUCTION</u>
CALL	function_name(params) as an expression
COMMENT	<u>EPRINTF</u>
CONTEXT	<u>SCOPE</u>
Control-C	STOP
COPROC	-
CREGDEF	Edit board file
CREGISTERS	@regname as an expression.
CWRITE	<u>SETREG</u>

Table C-1 armsd commands and equivalents (continued)

armsd commands	RealView Debugger commands
EXAMINE	DUMP
FIND	SEARCH
FPREGISTERS	@regname for each register as an expression
GETFILE	READFILE
GO	GO
HELP	DCOMMANDS HELP
IN	DOWN
ISTEP	STEPINSTR STEPOINSTR
LANGUAGE	-
LET	SETREG CEXPRESSION
LIST	DISASSEMBLE PRINTDSM MODE DUMP
LOAD	LOAD
LOG	LOG
LSYM	PRINTSYMBOLS
!value_memoryaddress = data_value	CEXPRESSION
OBEY	INCLUDE
OUT	UP
PAUSE	PAUSE
PRINT	CEXPRESSION PRINTF PRINTVALUE
PROFCLEAR	-

Table C-1 armsd commands and equivalents (continued)

armsd commands	RealView Debugger commands
PROFOFF	-
PROFON	-
PROFWRITE	-
PUTFILE	<u>WRITEFILE</u>
QUIT	QUIT
READSYMS	<u>LOAD</u> /NS
REGISTERS	@ <i>regname</i> for each register as an expression
RELOAD	<u>RELOAD</u>
RETURN	-
STEP	<u>S</u> TEPLINE STEP0
SYMBOLS	<u>E</u> XPAND <u>P</u> RINTSYMBOLS
TYPE	<u>L</u> IST
UNBREAK	<u>C</u> LEARBREAK
UNWATCH	<u>C</u> LEARBREAK
VARIABLE	<u>P</u> RINTTYPE
WATCH	<u>B</u> REAKACCESS <u>B</u> REAKREAD <u>B</u> REAKWRITE
WHERE	<u>C</u> ONTEXT
WHILE	<u>W</u> HILE statement (in macro or include file).

C.2 Converting legacy armsd scripts to RealView Debugger format

This section describes how you can convert legacy armsd scripts to the RealView Debugger format. The RealView Developer Suite product includes the program `armsd2rvd` to help with this process.

C.2.1 `armsd2rvd` command syntax

The command has the syntax:

```
armsd2rvd [-Vsn] [-A] -I infile.asd [-O rvdfile.txt]
```

where:

- I *infile.asd* The name of the existing script file. If you do not specify the input file, the command prints out a usage message. There must be a space between the file option and the filename.
- O *rvdfile.txt* The name of the new RealView Debugger script file. If you do not specify this, the converted script is written to the standard output, normally the terminal. There must be a space between the file option and the filename.
- A Include input command lines as comments in the output file.
- Vsn Print the version number on the standard output and exit.

You must quote filenames if they contain space characters. The filename extension used for the input or the output files is not predetermined by these programs.

Note

This command help you convert a script file to a RealView Debugger include file, but because of the different facilities provided by armsd, and the very different command set implemented, you must check the resulting file for commands that the script cannot convert.

In particular, armsd profiling commands are not converted because the RealView Debugger equivalents are too different.

See *Comparison of RealView Debugger and armsd commands* on page C-2 for a comparison of the commands available in armsd.

Appendix D

Mapping of Menu Options to Toolbar Buttons

This appendix shows the menu options that are equivalent to toolbar buttons.

D.1 Main menu to toolbar button mapping

Table D-1 list the main menu options that have equivalent toolbar buttons.

Table D-1 Main menu to toolbar button mapping

Main menu option	Toolbar button
File → New	
File → Open...	
File → Save	
File → Print...	
Edit → Cut	
Edit → Copy	
Edit → Paste	
Debug → Run	
Debug → Stop Execution	
Debug → Step Into	
Debug → Step Over (next)	
Debug → Run until Return	
Debug → Run to Cursor	
Target → Load Image...	
Target → Reload Image to Target	

Table D-1 Main menu to toolbar button mapping (continued)

Main menu option	Toolbar button
Debug → Set PC to Entry Point	
Target → Connect to Target...	
Target → Disconnect	
Build → Build...	
Build → Compile	
Build → Stop Build/Find	

Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

Address breakpoint A type of breakpoint.

See also Breakpoint.

ADS *See* ARM Developer Suite.

Angel Angel is a software debug monitor that runs on the target and enables you to debug applications running on ARM-based hardware. Angel is commonly used where a JTAG emulator is not available.

ARM Developer Suite (ADS)

A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors. ADS is superseded by RealView Developer Suite (RVDS).

See also RealView Developer Suite.

Backtracing *See* Call Stack.

Big-endian	Memory organization where the least significant byte of a word is at the highest address and the most significant byte is at the lowest address in the word. <i>See also</i> Little-endian.
Board	RealView Debugger uses the term <i>board</i> to refer to a target processor, memory, peripherals, and debugger connection method.
Board file	In RealView Debugger, the <i>board file</i> is the top-level configuration file, normally called <i>rvdebug.brd</i> , that references one or more other files.
Breakpoint	A user defined point at which execution stops in order that a debugger can examine the state of memory and registers. <i>See also</i> Hardware breakpoint and Software breakpoint.
Call Stack	This is a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.
Conditional breakpoint	A breakpoint that halts execution when a particular condition becomes True. The condition normally references the values of program variables that are in scope at the breakpoint location.
Context menu	<i>See</i> Pop-up menu.
Core module	In the context of Integrator, an add-on development board that contains an ARM processor and local memory. Core modules can run stand-alone, or can be stacked onto Integrator motherboards. <i>See also</i> Integrator.
DCC	<i>See</i> Debug Communications Channel.
Debug Agent (DA)	The Debug Agent resides on the target to provide target-side support for <i>Running System Debug</i> (RSD). The Debug Agent can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the <i>debug communications channel</i> (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state. <i>See also</i> Running System Debug.
Debug Communications Channel (DCC)	A debug communications channel enables data to be passed between RealView Debugger and the EmbeddedICE logic on the target using the JTAG interface, without stopping the program flow or entering debug state.

Deprecated	A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features are to be removed in future versions of the product.
Digital Signal Processor (DSP)	DSPs are special processors designed to execute repetitive, maths-intensive algorithms. Embedded applications might use both ARM processor cores and DSPs.
Doubleword	A 64-bit unit of information.
DSP	<i>See</i> Digital Signal Processor.
Embedded Trace Macrocell (ETM)	A block of logic, embedded in the hardware, that is connected to the address, data, and status signals of the processor. It broadcasts branch addresses, and data and status information in a compressed protocol through the trace port. It contains the resources used to trigger and filter the trace output.
EmbeddedICE logic	The EmbeddedICE logic is an on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface. <i>See also</i> IEEE1149.1.
ETM	<i>See</i> Embedded Trace Macrocell.
Execution vehicle	Part of the debug target interface, execution vehicles process requests from the client tools to the target.
Halfword	A 16-bit unit of information.
Halted System Debug (HSD)	Usually used for RTOS aware debugging, <i>Halted System Debug</i> (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. With the target stopped, the debugger presents RTOS information to you by reading and interpreting target memory. <i>See also</i> Running System Debug.
Hardware breakpoint	A breakpoint that is implemented using non-intrusive additional hardware. Hardware breakpoints are the only method of halting execution when the location is in <i>Read Only Memory</i> (ROM). Using a hardware breakpoint often results in the processor halting completely. This is usually undesirable for a real-time system. <i>See also</i> Breakpoint and Software breakpoint.
HSD	<i>See</i> Halted System Debug.

IEEE 1149.1	The IEEE Standard that defines TAP. Commonly (but incorrectly) referred to as JTAG. <i>See also</i> Test Access Port
Integrator	A range of ARM hardware development platforms. <i>Core modules</i> are available that contain the processor and local memory.
Joint Test Action Group (JTAG)	An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a <i>Joint Test Action Group</i> (JTAG) interface port to communicate with processors. For more information see IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).
JTAG	<i>See</i> Joint Test Action Group.
JTAG interface unit	A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the EmbeddedICE logic and the ETM.
Little-endian	Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word. <i>See also</i> Big-endian.
Multi-ICE	A JTAG-based tool for debugging embedded systems.
Pop-up menu	Also known as <i>Context menu</i> . A menu that is displayed temporarily, offering options relevant to your current situation. Obtainable in most RealView Debugger windows or panes by right-clicking with the mouse pointer inside the window. In some windows the pop-up menu can vary according to the line the mouse pointer is on and the tabbed page that is currently selected.
Processor core	The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.
RDI	<i>See</i> Remote Debug Interface.
RealView ARMulator ISS (RVISS)	The most recent version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite. It communicates with a debug target using RV-msg, through the RealView Connection Broker interface, and RDI. <i>See also</i> RDI and RealView Connection Broker.

RealView Compilation Tools (RVCT)

RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of *RISC* processors.

RealView Connection Broker

RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system. It also enables you to make multiple connections to the simulator.

See also RealView ARMulator ISS.

RealView Debugger Trace

Part of the RealView Debugger product that extends the debugging capability with the addition of real-time program and data tracing. It is available from the Code window.

RealView ICE (RVI) A JTAG-based debug solution to debug software running on ARM processors.

Remote Debug Interface (RDI)

The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a simulator running on the host (for example, RealView ARMulator ISS)
- a debug monitor running on hardware that is based on an ARM core accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, RealView ICE or Multi-ICE).

Remote_A

Remote_A is a software protocol converter and configuration interface. It converts between the RDI 1.5 software interface of a debugger and the Angel Debug Protocol used by Angel targets. It can communicate over a serial or Ethernet interface.

RSD

See Running System Debug.

RTOS

Real Time Operating System.

Running System Debug (RSD)

Used for RTOS aware debugging, *Running System Debug* (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a *Debug Agent* (DA) that resides on the target. The Debug Agent is scheduled along with other tasks in the system.

See also Debug Agent and Halted System Debug.

RVCT

See RealView Compilation Tools.

RVISS

See RealView ARMulator ISS.

Scan chain	A scan chain is made up of serially-connected devices that implement boundary-scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain. Processors might contain several shift registers to enable you to access selected parts of the device.
Scope	The range within which it is valid to access such items as a variable or a function.
Semihosting	A mechanism whereby I/O requests made in the application code are communicated to the host system, rather than being executed on the target.
Simulator	A simulator executes non-native instructions in software (simulating a core).
Software breakpoint	A <i>breakpoint</i> that is implemented by replacing an instruction in memory with one that causes the processor to take exceptional action. Because instruction memory must be altered software breakpoints cannot be used where instructions are stored in read-only memory. Using software breakpoints can enable interrupt processing to continue during the breakpoint, making them more suitable for use in real-time systems. <i>See also</i> Breakpoint and Hardware breakpoint.
Software Interrupt (SWI)	An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.
SWI	<i>See</i> Software Interrupt.
TAP	<i>See</i> Test Access Port.
TAP Controller	Logic on a device which enables access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1. <i>See also</i> Test Access Port and IEEE1149.1.
Target	The target hardware, including processor, memory, and peripherals, real or simulated, on which the target application is running.
Target vehicle	Target vehicles provide RealView Debugger with a standard interface to disparate targets so that the debugger can connect easily to new target types without having to make changes to the debugger core software.
Target Vehicle Server (TVS)	Essentially the debugger itself, this contains the basic debugging functionality. TVS contains the run control, base multitasking support, much of the command handling, and target knowledge, such as memory mapping, lists, rule processing, board file and .bcd files, and data structures to track the target environment.
Test Access Port (TAP)	The port used to access the TAP Controller for a given device. Comprises TCK , TMS , TDI , TDO , and nTRST (optional).

Tracepoint	A tracepoint can be a line of source code, a line of assembly code, or a memory address. In RealView Debugger, you can set a variety of tracepoints to determine exactly what program information is traced.
Tracing	The real-time recording of processor activity (including instructions and data accesses) that occurs during program execution. Trace information can be stored either in a trace buffer of a processor, or in an external trace hardware unit. Captured trace information is returned to the Analysis window in RealView Debugger where it can be analyzed to help identify a defect in program code.
Trigger	In the context of breakpoints, a trigger is the action of noticing that the breakpoint has been reached by the target and that any associated conditions are met. In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.
TVS	<i>See</i> Target Vehicle Server.
Watch	A watch is a variable or expression that you require the debugger to display at every step or breakpoint so that you can see how its value changes. The Watch pane is part of the RealView Debugger Code window that displays the watches you have defined.
Watchpoint	In RealView Debugger, this is a hardware breakpoint.
Word	A 32-bit unit of information.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

- About this book x
- Access-provider connection 5-10
- ARM-ARM-NW vehicle 5-10
- armsd
 - moving to RealView Debugger C-1
- armsd scripts 7-3
 - converting to RealView Debugger
 - format C-5
- Audience, intended x
- Autobinding 6-6
- Automatic operations
 - setting up 6-7
 - summary 6-6
- Auto-projects
 - creating for images 5-42
 - properties 5-42
 - see also* Projects
- AXD
 - moving to RealView Debugger B-1
- AXD scripts 7-3

- converting to RealView Debugger
 - format B-16

B

- Batched commands 7-6
 - adding comments 7-8
 - comments and journal files 7-9
 - comments and log files 7-9
 - rules for comments 7-8
 - using QUIT command 7-9
- Binding 6-6
 - autobinding 6-6
 - default binding 6-7
 - specific device binding 6-6
- Board file 3-4
- Board/Chip Definition file 3-4
- Book, about this x
- Breakpoints 5-30
 - conditional 5-32
 - default 5-32
 - hardware 5-31

- software 5-31
- unconditional 5-32
- Build target configurations 6-3

C

- Call Stack** tab 4-7, 5-45
- Cancel command 4-18
- Close down
 - connected 5-55
 - disconnected 5-54
- Cmd** tab 5-15, 5-17, 5-20, 5-27, 5-36, 5-44, 7-2, 7-3, 7-5
- Code views 4-6, 5-20, 5-30
 - disassembly tab 5-20, 5-30
- Dsm** tab 5-20, 5-30
- source code tab 5-20, 5-30
- Src** tab 5-20, 5-30
- Code window 4-21
 - attachment 4-5
 - Button toolbars 4-17
 - button toolbars 4-11

- Color Box 4-12
 - components 4-3
 - Cursor location field 4-13
 - customizing toolbars 4-20
 - debug views 4-8
 - default 4-3
 - default panes 4-8
 - default window 4-5
 - File Editor pane 4-6
 - Find toolbar 4-19
 - focus control 4-24
 - in-place editing 4-24, 5-18
 - JOU 4-13
 - LOG 4-13
 - Main menu 4-14
 - pane controls 4-9
 - panes 4-8
 - Processor status 4-13, 5-30
 - Resource Viewer 5-22
 - Status line 4-12
 - STDIOLog 4-13
 - title bar 4-4
 - toolbar 4-11, 4-17
 - Color Box 4-12
 - Command
 - syntax 7-5
 - Command history 7-5
 - Command line
 - b option 7-6
 - debugger options on 7-6
 - inc option 7-6
 - see also* Headless debugger
 - Comments
 - on documentation xv
 - on RealView Debugger xv
 - Concepts
 - connection 3-2
 - debug target 3-2
 - DSP 3-2
 - multiprocessor access 3-2
 - multithreading 3-3
 - RTOS 3-2
 - single connection access 3-2
 - Concepts and terminology 3-2
 - see also* Glossary
 - Conditional breakpoints 5-32
 - Conn** tab 5-23, 5-25
 - in Resource Viewer 5-25
 - Connect mode 5-13
 - Connect** tab 5-10, 5-11
 - Connecting 5-11, 5-12
 - and close down 5-55
 - Connection
 - concept 3-2
 - Connection Broker 3-5
 - Local mode 3-5
 - Remote mode 3-5
 - Connection Control window
 - access-provider connection 5-10
 - contents 5-10
 - displaying 5-9
 - endpoint connections 5-10
 - managing connections 5-12
 - target execution vehicle 5-10
 - Connections
 - cycling through 4-18, 5-25
 - interdependence with projects 6-6
 - Container projects 6-2
 - Core** tab 5-39, 5-40
 - Custom projects 6-2
 - Custom windows
 - using 4-15
 - Custom Windows** option 4-15
 - Customizing
 - toolbars 4-20
 - Cycle Connections** button
 - Code window 4-18
 - Resource Viewer window 5-25
 - Cycle Threads** button 4-18
 - CycleCount** tab 5-39, B-2, B-3
- ## D
- DA
 - Debug Agent 5-33
 - Data Navigator pane 5-46
 - Functions** tab 5-47
 - Modules** tab 5-47
 - DCOMMANDS command 7-4
 - Debug Agent 5-33
 - Debug** tab 5-17, B-2, B-3, B-5
 - Debug target
 - concept 3-2
 - configuration 5-12
 - Connect mode 5-13
 - connecting 5-11, 5-12
 - Disconnect mode 5-52
 - disconnecting 5-52
 - displaying target status 5-30
 - execution vehicles 3-5
 - interface 3-5
 - Multi-ICE 3-5
 - Processor status 5-30
 - RealView ARMulator ISS 2-10, 3-5
 - RealView ICE 3-5
 - reconnecting using workspace 5-56
 - Debug target interface 3-5
 - Debugger internals
 - Register pane 5-39
 - Debugging environment 3-4
 - Debugging mode 3-7
 - Default binding 6-7
 - Default breakpoints 5-32
 - Default_encoding setting for
 - internationalization 5-6
 - Desktop
 - adding panes 4-9
 - attaching windows 4-5
 - bottom pane 4-7
 - Build** menu 4-15
 - Call Stack pane 4-7
 - Code window 4-3
 - Color Box 4-12
 - Cursor location field 4-13
 - customizing toolbars 4-20
 - Debug** menu 4-15
 - Edit** menu 4-14
 - File Editor pane 4-6
 - File** menu 4-14
 - Find** menu 4-14
 - Find toolbar 4-19
 - floating toolbars 4-11
 - focus 4-24
 - Help** menu 4-16
 - hiding toolbars 4-11
 - in-place editing 4-24, 5-18
 - JOU 4-13
 - LOG 4-13
 - main menu 4-14
 - Memory pane 4-7
 - middle row pane 4-7
 - Pane Control** menu 4-9
 - pane controls 4-9
 - pane gripper bar 4-10
 - Pane** menu 4-10
 - pane reposition control 4-10

- pane resize control 4-10
- Process Control pane 4-6
- Processor status 4-13
- Project** menu 4-15
- scroll bars 4-25
- Scroll** menu 4-25
- side pane 4-6
- splash screen 4-2
- Status line 4-12
- STDIOLog 4-13
- substituting panes 4-8, 4-9
- tabs 4-25
- Target** menu 4-15
- toolbar 4-11, 4-17
- Tools** menu 4-15
- View** menu 4-14
- viewing toolbars 4-11
- Watch pane 4-7
- DHELP alias 7-4
- Dialog box
 - Interactive Register Settings 5-16
 - Set Address/Data Break/Tracepoint 5-34
- Directories
 - default home A-5
 - default settings A-3
 - project base 5-3
 - \etc A-3
 - \Examples x
 - \home A-5
- Disassembly view 5-20, 5-30
- Disconnect mode 5-52
- Disconnecting 5-52
 - and close down 5-54
- Documentation
 - DynaText 2-17
 - online 2-17
 - PDF 2-17
 - structure of this book xi
 - using the suite 1-3
- Documentation feedback xv
- Dsm** tab 4-6, 5-30, 5-44, 5-47
- DSP
 - concept 3-2

E

- Encoding format

- for searches 4-19, 5-7
- global setting 5-7
- see also* Internationalization
- setting in Call Stack pane 5-8
- setting in File Editor pane 4-19, 5-7
- setting in Output pane 5-8
- setting in Watch pane 5-8
- workspace setting 5-7
- Endpoint connections 5-10
- Enquiries xv
- Entering commands directly 7-5
- Execution vehicle
 - see* Debug target
- Exiting
 - RealView Debugger 5-54
- Expression pointer 5-46
- Extended Target Visibility 7-7

F

- Favorites
 - persistence info 5-50
- Feedback
 - on documentation xv
 - on RealView Debugger xv
- File Editor
 - File Editor pane 4-6
- File** menu
 - in Resource Viewer 5-23
- Files
 - armul.var A-3
 - board file 3-4
 - Board/Chip Definition 3-4
 - configuration A-3
 - genmake.loc A-3
 - genmake_ARM_ADS.loc A-3
 - gen_arx.mk A-3
 - installed in \etc A-3
 - installed in \home A-5
 - pARM.prc A-3
 - rvdebug.brd A-3
 - rvdebug.tp1 A-3
 - startup.mk A-3
 - targ_ARM.aco A-3
 - template.spr A-4
 - *.auc A-5
 - *.aws 5-49, A-5
 - *.axf 5-19

- *.bak A-6
- *.bcd 3-4, A-4, A-6
- *.brd 3-4, A-5
- *.cnf A-6
- *.ini A-5
- *.jtg A-4
- *.loc A-5
- *.rbe A-6
- *.rvc A-6
- *.sav 5-50, A-5
- *.sig A-5
- *.stp A-4

Focus

- in Code window 4-24
- Font recommendations
 - Chinese 5-5
 - Japanese 5-5
- Frame pointer 5-46
- Functions** tab
 - Data Navigator pane 5-47

G

- gcc support 2-3
- Glossary Glossary-1

H

- Halted System Debug
 - see* HSD
- Headless debugger 7-6, 7-10
- Help on Commands 7-4
- Help, online 7-4
- History list
 - saving 5-50
- HSD
 - defined 5-32
 - switching to RSD 5-33

I

- Images
 - loading 5-19
 - loading directly 5-19
 - loading from a project 5-20

- loading multi-image applications 5-21
 - reloading 5-21
 - unloading 5-48
 - Images** tab 5-47
 - INCLUDE** command 7-7
 - Include** files 7-6
 - adding comments 7-8
 - comments and journal files 7-9
 - comments and log files 7-9
 - rules for comments 7-8
 - using **QUIT** command in 7-9
 - In-place editing 4-24, 5-18
 - focus 4-25
 - Intended audience x
 - Interactive command entry 7-5
 - Interactive Register Settings dialog 5-16
 - Internationalization
 - configuring 5-6
 - `DefaultEncoding` setting 5-7
 - Enabled setting 5-6, 5-7
 - enabling 5-6, 5-7
 - font recommendations 5-5
 - Language setting 5-6
 - see also* Encoding format
 - Invoking a batch file 7-6
- ## L
- Language setting for Internationalization 5-6
 - Library projects 6-2
 - Line numbers
 - setting breakpoints 5-34
 - turning on 5-19, 5-29
 - Loading
 - images 5-19
 - Localization
 - see* Internationalization
 - Locals** tab 4-7, 5-45
- ## M
- Maximum command length 7-5
 - Memory 5-42
 - for ARM processors 5-15
 - Menus
 - Build** 4-15
 - Debug** 4-15
 - Edit** 4-14
 - File** 4-14
 - Find** 4-14
 - Help** 4-16
 - Pane** 4-10
 - Pane Control** 4-9
 - Project** 4-15
 - Target** 4-15
 - Tools** 4-15
 - View** 4-14
 - Workspace** 5-49
 - Mode
 - debugging 3-7
 - Modules** tab
 - Data Navigator pane 5-47
 - Multibyte characters
 - see* Encoding format
 - see* Internationalization
 - Multi-ICE 3-5
 - Multiprocessor
 - viewing resources 5-25
 - Multiprocessor access
 - concept 3-2
 - Multithreading
 - concept 3-3
- ## N
- Network Broker 3-5
 - Numbers
 - turn on lines 5-19, 5-29
- ## O
- Online help 7-4
- ## P
- Panes
 - adding 4-9
 - bottom 4-7
 - Break/Tracepoints 5-37
 - Call Stack 4-7
 - Data Navigator 5-46
 - File Editor 4-6
 - focus 4-24
 - hidden 4-10
 - making a hidden pane visible 4-10
 - Memory 4-7, 5-42, 5-46
 - middle row 4-7
 - pane controls 4-9
 - Process Control 4-6
 - Register 5-17
 - side 4-6
 - Stack 5-45
 - substituting 4-8
 - Symbol Browser 5-47
 - Watch 4-7, 5-38
 - PC 5-19
 - Persistence info 3-6, 5-49, 5-51
 - favorites 5-50
 - files A-5
 - Problem solving xv
 - Process Control pane
 - type ahead 5-41
 - Process** tab 5-41, 5-42
 - Processor status 5-30
 - Product feedback xv
 - Program Counter 5-19
 - Projects
 - associated images 5-42
 - autobinding 6-6
 - automatic operations 6-6
 - auto-projects 5-42
 - binding 6-6
 - bound 4-4
 - build target configurations 6-3
 - Container 6-2
 - Custom 6-2
 - default binding 6-7
 - getting started 6-1
 - interdependence with connections 6-6
 - Library 6-2
 - limitations of 6-5
 - specific device binding 6-6
 - Standard 6-2
 - unbound 4-4

Q

Queries xv

R

Real Time Operating Systems 3-2

RealView ARMulator ISS 2-10, 3-5

breakpoints 5-35

connecting 5-12

debugger internals 5-39

in Connection Control window 5-12

in examples x

memory 5-15

setting top of stack 5-15

setting top_of_memory 5-15

RealView Connection Broker 3-5

RealView Debugger

closing down 5-54, 5-55

Code window 4-3

components 3-4

concepts 3-2

connecting to a target 5-11

debug target interface 3-5

debugging mode 3-7

desktop 4-2

environment 3-4

exiting 5-54

for armsd users C-1

for AXD users B-1

gcc support 2-3

image loading 5-19

image reloading 5-21

image unloading 5-48

introduction 3-2

persistence info 3-6

projects 6-1

starting 5-4

stopping 5-54

target configuration 5-12

tutorial 5-2

TVS 3-4

using the documentation 1-3

window attachment 4-5

RealView ICE 3-5

debugger internals 5-39

RealView Network Broker 3-5

RealView Simulator Broker 3-5

Registers 5-39

banks 5-40

changing contents 5-16, 5-18

Reloading

images 5-21

Resource Viewer 5-22

RSD

defined 5-33

RSD menu (Resource Viewer) 5-24

switching to HSD 5-33

RSD menu

in Resource Viewer 5-24

RTOS 3-2

concept 3-2

see also HSD

see also RSD

threads 3-3

viewing resources 5-25

Running System Debug

see RSD

S

Save

session 5-49

Settings options 5-51

Scripts 7-6

adding comments 7-8

comments and journal files 7-9

comments and log files 7-9

rules for comments 7-8

using QUIT command in 7-9

Scrolling

menu 4-25

Semihost tab B-2, B-5

Session

history list 5-50

saving 5-49

Settings options 5-51

startup file 5-50

workspace 5-49

Set Address/Data Break/Tracepoint

dialog 5-34

Simulator Broker 3-5

Single connection access

concept 3-2

Software Problem Report

see SPR

Source file tab 5-30

Source view 5-20, 5-30

Specific device binding 6-6

SPR xv, 4-16

Src tab 4-6, 5-20, 5-30, 5-33, 5-44

Stack

EP 5-46

for RealView ARMulator ISS 5-15

FP 5-46

SP 5-46

Stack pointer 5-46

default 5-46

Standard projects 6-2

Starting

RealView Debugger 5-4

Startup file

saving 5-50

Statics tab 4-7, 5-45

StdIO tab 5-27, 7-15

Structure of this book xi

Symbol Browser pane 5-47

Synch tab 5-11

Syntax of commands 7-5

T

Tabbed pages 4-25

Tabs 4-25

list of available 4-25

red text 4-25

right-click 4-25

right-click on blank area 4-25

right-click on scroll arrow 4-25

using 4-25

Target

see Debug target

Target execution vehicle 5-10

Target Vehicle Server 3-4

Terminology Glossary-1

Terminology and concepts 3-2

Text encoding

see Encoding format 5-7

This tab 4-7

Threads 3-3

see also HSD

see also RSD

Threads tab 4-6

Toolbars

- Build 4-19
 - Command cancel 4-18
 - Connect 4-18
 - Connection** button
 - in Resource Viewer 5-25
 - Context controls 4-18
 - customizing 4-20
 - Cycle Threads** button 4-18
 - Debug 4-17
 - Edit 4-17
 - Encoding** field 4-19
 - Execution controls 4-17
 - File 4-17
 - File** field 4-19
 - Find 4-19
 - Find** field 4-19
 - Image 4-18
 - Line** field 4-19
 - Source control** button 4-19
 - Stepping controls 4-17
 - Tooltip Evaluation 5-44
 - Tooltips 4-17
 - top_of_memory
 - for RealView ARMulator ISS 5-15
 - Tutorial
 - batched commands 7-6
 - breakpoints 5-30
 - changing memory contents 5-43
 - clearing breakpoints 5-37
 - conditional breakpoints 5-34
 - debugger internals 5-39
 - default breakpoints 5-33
 - deleting 5-57
 - displaying memory 5-42
 - displaying registers 5-39
 - entering commands in the GUI 7-5
 - headless debugger 7-10
 - how to use 5-2
 - managing breakpoints 5-36
 - process control 5-40
 - setting up a project 6-1
 - setting watches 5-37
 - using the call stack 5-44
 - using the Data Navigator pane 5-46
 - using the stack 5-45
 - using the Symbol Browser pane 5-47
 - using tooltip evaluation 5-44
 - using tooltips 4-17
 - TVS 3-4
- ## U
- Unconditional breakpoints 5-32
 - Unloading
 - images 5-48
 - Using the CLI 7-3
- ## V
- Vehicle
 - ARM-ARM-NW 5-10
 - View** menu
 - in Resource Viewer 5-23
 - Views
 - code 5-20, 5-30
- ## W
- Watch** tabs 4-7
 - Watches 5-37
 - Windows
 - attachment 4-4, 4-5
 - focus 4-24
 - Resource Viewer 5-22
 - Workspace
 - connection details 5-52, 5-54, 5-56
 - saving 5-49