

RealView® Debugger

Version 4.1

User Guide

ARM®

RealView Debugger

User Guide

Copyright © 2002-2010 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History			
Date	Issue	Confidentiality	Change
April 2002	A	Non-Confidential	Release v1.5
September 2002	B	Non-Confidential	Release v1.6
February 2003	C	Non-Confidential	Release v1.6.1
September 2003	D	Non-Confidential	Release v1.6.1 for RealView Developer Suite v2.0
January 2004	E	Non-Confidential	Release v1.7 for RealView Developer Suite v2.1
December 2004	F	Non-Confidential	Release v1.8 for RealView Developer Suite v2.2
May 2005	G	Non-Confidential	Release v1.8 SP1 for RealView Developer Suite v2.2 SP1
March 2006	H	Non-Confidential	Release v3.0 for RealView Development Suite v3.0
March 2007	I	Non-Confidential	Release v3.1 for RealView Development Suite v3.1
September 2008	J	Non-Confidential	Release 4.0 for RealView Development Suite v4.0
27 March 2009	K	Non-Confidential	Release v4.0 SP1 for RealView Development Suite v4.0
28 May 2010	L	Non-Confidential	Release 4.1 for RealView Development Suite v4.1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. 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.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

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 User Guide

Preface

About this book	x
Feedback	xv

Chapter 1

RealView Debugger Features

1.1 Overview of RealView Debugger windows and views	1-2
1.2 Target connection	1-23
1.3 Image and binary loading	1-26
1.4 TrustZone technology support	1-28
1.5 Multiprocessor debugging	1-29
1.6 Execution control	1-31
1.7 Memory mapping	1-32
1.8 Execution context and scope	1-34
1.9 Breakpoints in RealView Debugger	1-36
1.10 Examining the target execution environment	1-38
1.11 Altering the target execution environment	1-40
1.12 Command scripts	1-42
1.13 Macros	1-43
1.14 Log and journal files	1-44
1.15 Editing source files	1-45
1.16 Searching source files	1-46

Chapter 2

The RealView Debugger Environment

2.1 Starting RealView Debugger from the command line	2-2
2.2 Starting RealView Debugger after installing other components	2-7
2.3 Setting user-defined environment variables	2-8
2.4 Redefining the RealView Debugger directories	2-9

Chapter 3**Target Connection**

3.1	About target connection	3-2
3.2	About creating a Debug Configuration	3-8
3.3	Changing the name of a Debug Configuration	3-17
3.4	Copying an existing Debug Configuration	3-18
3.5	Deleting a Debug Configuration	3-19
3.6	Customizing a Debug Configuration	3-20
3.7	Connecting to a target	3-27
3.8	Showing the trace components in the Connect to Target window	3-35
3.9	Viewing information about the target topology	3-36
3.10	Setting top of memory for the current debugging session	3-38
3.11	Viewing connection details	3-40
3.12	Connecting to a target on startup	3-42
3.13	Connecting to a target using different modes	3-43
3.14	Connecting to multiple targets	3-46
3.15	Connecting to all targets for a specific Debug Configuration	3-48
3.16	Changing the current target connection	3-50
3.17	Disconnecting from a target	3-52
3.18	Disconnecting from a target using different modes	3-54
3.19	Disconnecting from multiple targets	3-56
3.20	Disconnecting from all targets for a specific Debug Configuration	3-58
3.21	Storing connections when exiting RealView Debugger	3-59
3.22	Troubleshooting target connection problems	3-60

Chapter 4**Loading Images and binaries**

4.1	About loading images and binaries	4-2
4.2	Loading an executable image	4-4
4.3	Viewing image details	4-9
4.4	Loading a binary	4-12
4.5	Loading multiple images to the same target	4-14
4.6	Loading symbols only for an image	4-16
4.7	Replacing the currently loaded image	4-17
4.8	Loading an executable image on startup	4-18
4.9	Unloading an image	4-20
4.10	Deleting the process details for an unloaded image	4-21
4.11	Reloading an image	4-22
4.12	Reloading a binary	4-23
4.13	Changing the format of the disassembly view	4-24
4.14	Interleaving source in the disassembly view	4-25
4.15	Opening source files for a loaded image	4-26
4.16	Saving and closing source files	4-28
4.17	Hiding line numbers for opened source files	4-29
4.18	Adding source file search paths for a loaded image	4-31
4.19	Autoconfiguring search rules for locating source files	4-34

Chapter 5**Navigating the Source and Disassembly Views**

5.1	About navigating the source and disassembly views	5-2
5.2	Viewing the selected location in the opposite code view	5-3
5.3	Locating the lowest address in memory of a module	5-4
5.4	Locating the line of code using a symbol in the source view	5-6
5.5	Locating the address of a label in the disassembly view	5-7
5.6	Locating the destination of a branch instruction	5-8
5.7	Locating a function	5-9

Chapter 6**Writing Binaries to Flash**

6.1	About writing binaries to Flash	6-2
6.2	Writing a binary to Flash	6-4
6.3	Writing to specific locations in Flash memory	6-6
6.4	Viewing information about the Flash memory	6-8

	6.5	Operations available when writing to Flash	6-10
Chapter 7	Debugging Multiprocessor Applications		
7.1	About debugging multiprocessor applications	7-2	
7.2	Displaying multiple Code windows	7-8	
7.3	Attaching a Code window to a connection	7-10	
7.4	Unattaching a Code window from a connection	7-11	
7.5	Using DSTREAM or RealView ICE for multiprocessor debugging	7-12	
7.6	Synchronizing multiple processors	7-13	
7.7	Setting up software cross-triggering	7-17	
7.8	Setting up hardware cross-triggering	7-20	
7.9	Configuring embedded cross-triggering	7-25	
7.10	Configuring CoreSight embedded cross-triggering	7-27	
7.11	Sharing resources between multiple targets	7-28	
Chapter 8	Executing Images		
8.1	About image execution	8-2	
8.2	Starting and stopping image execution	8-4	
8.3	Running an image to a specific point	8-7	
8.4	Stepping by lines of source code	8-12	
8.5	Stepping by instructions	8-16	
8.6	Stepping until a user-specified condition is met	8-20	
8.7	Resetting your target processor	8-22	
Chapter 9	Mapping Target Memory		
9.1	About mapping target memory	9-2	
9.2	Setting the default access type for unmapped memory regions	9-5	
9.3	Enabling memory mapping	9-6	
9.4	Viewing the memory map	9-8	
9.5	Setting up a temporary memory map	9-12	
9.6	Setting up a memory map	9-15	
9.7	Creating a temporary memory map entry	9-18	
9.8	Editing a memory map entry	9-20	
9.9	Updating the memory map	9-22	
9.10	Deleting memory map blocks	9-23	
9.11	Generating linker command files for non-ARM targets	9-24	
Chapter 10	Changing the Execution Context		
10.1	About changing the execution context	10-2	
10.2	Changing scope to the PC	10-5	
10.3	Displaying the current execution context	10-6	
10.4	Resetting the PC to the image entry point	10-7	
10.5	Setting the PC to the address of an instruction or line of code	10-8	
10.6	Setting the PC to a function	10-9	
10.7	Changing scope to the code pointed to by a Call Stack entry	10-12	
Chapter 11	Setting Breakpoints		
11.1	About setting breakpoints	11-3	
11.2	Setting a simple breakpoint	11-13	
11.3	Setting an unconditional breakpoint with specific attributes	11-16	
11.4	Clearing breakpoints	11-18	
11.5	Viewing breakpoint information	11-20	
11.6	Disabling a breakpoint	11-22	
11.7	Enabling a breakpoint	11-23	
11.8	Editing a breakpoint	11-25	
11.9	Copying a breakpoint	11-28	
11.10	Finding a breakpoint in the code view	11-30	
11.11	Viewing the target hardware breakpoint support	11-31	
11.12	Setting breakpoints by dragging and dropping	11-32	

	11.13	Setting breakpoints on lines of source code	11-34
	11.14	Setting breakpoints on instructions	11-37
	11.15	Setting breakpoints on functions	11-39
	11.16	Setting breakpoints for memory accesses	11-46
	11.17	Setting breakpoints for location-specific data values	11-54
	11.18	Setting breakpoints for location-independent data values	11-59
	11.19	Forcing the size of a software breakpoint	11-62
	11.20	Chaining hardware breakpoints	11-63
	11.21	Specifying processor exceptions (global breakpoints)	11-65
	11.22	Setting breakpoints on custom memory mapped registers	11-67
	11.23	Setting breakpoints from the breakpoint history list	11-70
	11.24	Creating new breakpoint favorites	11-72
	11.25	Setting breakpoints from your Favorites List	11-74
Chapter 12	Controlling the Behavior of Breakpoints		
	12.1	About controlling the behavior of breakpoints	12-2
	12.2	Updating windows and views when a breakpoint activates	12-4
	12.3	Displaying user-defined messages when a breakpoint activates	12-7
	12.4	Setting the execution behavior for a breakpoint	12-9
	12.5	Setting breakpoints that test for hardware input triggers	12-10
	12.6	Setting a breakpoint that activates after a number of passes	12-13
	12.7	Resetting breakpoint pass counters	12-17
	12.8	Setting a breakpoint that depends on the result of an expression	12-18
	12.9	Setting a breakpoint that depends on the result of a macro	12-21
	12.10	Setting a breakpoint on a specific instance of a C++ class	12-24
	12.11	Example of breakpoint behavior	12-27
Chapter 13	Examining the Target Execution Environment		
	13.1	About examining the target execution environment	13-3
	13.2	Finding a function in your code	13-7
	13.3	Displaying function information from the Symbols view	13-10
	13.4	Displaying the list of variables in an image	13-11
	13.5	Viewing variables for the current context	13-14
	13.6	Displaying information for a variable	13-19
	13.7	Viewing C++ classes	13-22
	13.8	Viewing registers	13-25
	13.9	Viewing semihosting controls for DSTREAM or RealView ICE JTAG connections	13-35
	13.10	Viewing semihosting controls for RVISS targets	13-37
	13.11	Viewing memory contents	13-39
	13.12	MMU page tables views	13-46
	13.13	Managing the display of memory in the Memory view	13-50
	13.14	Viewing the Stack	13-58
	13.15	Viewing the Call Stack	13-63
	13.16	Setting watches	13-67
	13.17	Viewing watches	13-70
	13.18	Viewing statistics for RVISS targets	13-74
	13.19	Viewing the RVISS map related statistics in RealView Debugger	13-81
	13.20	Saving memory contents to a file	13-83
	13.21	Comparing target memory with the contents of a file	13-85
	13.22	Displaying information in a user-defined window	13-87
	13.23	Saving information to a user-defined file	13-90
	13.24	Displaying a list of open user-defined windows and files	13-93
Chapter 14	Altering the Target Execution Environment		
	14.1	About altering the target execution environment	14-2
	14.2	Changing the value of a register	14-3
	14.3	Changing memory contents	14-12
	14.4	Changing the data width on memory accesses	14-18
	14.5	Loading the contents of a file into memory	14-19

	14.6	Changing the stack pointer	14-21
	14.7	Changing the value of a watch	14-23
	14.8	Communicating with a target over DCC	14-24
Chapter 15	Debugging with Command Scripts		
	15.1	About debugging with command scripts	15-2
	15.2	Changing output buffering behavior	15-4
	15.3	Creating a log file for use as a command script	15-5
	15.4	Creating log and journal files at start-up	15-6
	15.5	Closing log and journal files	15-7
	15.6	Using macros in command scripts	15-8
	15.7	Running command scripts	15-9
	15.8	Creating a script that writes information to a user-defined window	15-12
	15.9	Creating a script that accesses a user-defined file	15-13
Chapter 16	Using Macros for Debugging		
	16.1	About using macros for debugging	16-2
	16.2	Creating a macro	16-4
	16.3	Loading user-defined macros	16-9
	16.4	Running a macro	16-12
	16.5	Editing a macro	16-14
	16.6	Copying a macro	16-16
	16.7	Viewing a macro	16-17
	16.8	Deleting a macro	16-18
	16.9	Using macros in combination with other commands	16-19
	16.10	Stopping execution of a macro	16-22
Chapter 17	Configuring Workspace Settings		
	17.1	About workspace settings	17-2
	17.2	Initializing the workspace	17-3
	17.3	Opening workspaces	17-4
	17.4	Closing workspaces	17-6
	17.5	Creating an empty workspace	17-8
	17.6	Saving workspaces	17-9
	17.7	Viewing workspace settings	17-11
	17.8	Configuring workspace settings	17-15
Appendix A	Workspace Settings Reference		
	A.1	DEBUGGER	A-2
	A.2	CODE	A-6
	A.3	ALL	A-7
	A.4	CONNECTION	A-14
	A.5	WINDOW	A-15
Appendix B	Configuration Files Reference		
	B.1	Overview	B-2
	B.2	Files in the default settings directory	B-3
	B.3	Files in the home directory	B-5
Appendix C	Moving from AXD to RealView Debugger		
	C.1	RealView Debugger configuration	C-2
	C.2	RealView Debugger operations	C-5
	C.3	Comparison of RealView Debugger and AXD commands	C-7
	C.4	Converting legacy AXD scripts to RealView Debugger format	C-11
Appendix D	Moving from armsd to RealView Debugger		
	D.1	RealView Debugger configuration	D-2
	D.2	Comparison of RealView Debugger and armsd commands	D-3

D.3	Converting legacy armsd scripts to RealView Debugger format	D-6
Appendix E		
RealView Debugger on Red Hat Linux		
E.1	About this Appendix	E-2
E.2	Getting more information	E-3
E.3	Changes to target configuration details	E-4
E.4	Changes to GUI and general user information	E-5

Preface

This preface introduces the *RealView® Debugger User Guide*. It contains the following sections:

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

About this book

This book describes how to use RealView Debugger to debug applications and images:

- a detailed description of how to use RealView Debugger to debug images using a range of debug targets, including examples
- a description of the built-in features of RealView Debugger, such as workspaces and macros
- appendixes containing reference information for the software developer.

Intended audience

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

This book includes appendixes that contains information for developers:

- using RealView Debugger on Red Hat Linux
- moving from *ARM eXtended Debugger* (AXD) or *ARM Symbolic Debugger* (armsd) to RealView Debugger.

Before you start

It is recommended that you read the *RealView Debugger Essentials Guide* before starting to use this book.

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 sample programs stored in the following directory:

install_directory\RVDS\Examples

In general, the examples in this book use *RealView ARMulator® ISS* (RVISS) to simulate an ARM architecture-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 *RealView Debugger Features*

This chapter gives an introduction to the RealView Debugger features.

Chapter 2 *The RealView Debugger Environment*

This chapter describes how to start RealView Debugger at the command line, and how to set up environment variables and directories to your requirements.

Chapter 3 Target Connection

This chapter describes how you connect to your target using the RealView Debugger Connect to Target window. It includes details of the context menus and how to connect to targets in specific ways.

Chapter 4 Loading Images and binaries

This chapter describes how to load an images and binaries ready for debugging and how to view image details.

Chapter 5 Navigating the Source and Disassembly Views

This chapter describes how to navigate to specific parts of your source code and the disassembly view.

Chapter 6 Writing Binaries to Flash

This chapter describes how to write binaries to Flash and to write to specific locations in Flash.

Chapter 7 Debugging Multiprocessor Applications

This chapter describes the features of RealView Debugger that enable you to debug multiprocessor applications and compare the behavior of different targets, for example two ARM processors.

Chapter 9 Mapping Target Memory

This chapter describes how to manage memory during a debugging session. It describes the Process Control view that contains a dynamic display of the current memory configuration.

Chapter 8 Executing Images

This chapter describes how to control image execution using the various running and stepping features.

Chapter 10 Changing the Execution Context

This chapter describes how to change the execution context during a debugging session.

Chapter 11 Setting Breakpoints

This chapter describes how to use breakpoints to control execution of your application program. This chapter contains a full description of breakpoint options in RealView Debugger.

Chapter 12 Controlling the Behavior of Breakpoints

This chapter describes how you can modify the behavior of breakpoints so that the activation of a breakpoint can be delayed and, when activated, what actions it performs.

Chapter 13 Examining the Target Execution Environment

This chapter describes how to monitor execution of your application program by setting watches, reading registers and tracking changes to memory contents.

Chapter 14 Altering the Target Execution Environment

This chapter describes how to change the execution of your application program by changing the values of watches, registers, and memory contents.

Chapter 15 *Debugging with Command Scripts*

This chapter describes how to use scripts to run RealView Debugger CLI commands to enable you to automate debugging operations.

Chapter 16 *Using Macros for Debugging*

This chapter describes how to create and use macros when working with RealView Debugger.

Chapter 17 *Configuring Workspace Settings*

RealView Debugger uses a workspace to enable you to configure your working environment and to maintain persistence information from one session to the next. You achieve this by using a workspace properties file and a global configuration file. This chapter describes the contents of these files and how to change your settings.

Appendices

Appendix A *Workspace Settings Reference*

This appendix describes how to set options to configure your working environment using RealView Debugger workspaces. This appendix must be read in association with Chapter 17 *Configuring Workspace Settings*.

Appendix B *Configuration Files Reference*

This appendix describes the files created when you install RealView Debugger v3.0. It describes where files are stored and what information each file contains.

Appendix C *Moving from AXD to RealView Debugger*

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

Appendix D *Moving from armsd to RealView Debugger*

This appendix is aimed at developers moving from armsd to RealView Debugger. This appendix describes how to convert scripts and provides a comparison of the armsd commands with equivalent RealView Debugger commands.

Appendix E *RealView Debugger on Red Hat Linux*

This appendix describes how certain features of RealView Debugger differ on Red Hat Linux, and contains corrections and additions to the documentation suite.

Typographical conventions

The typographical conventions are:

<i>italic</i>	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes 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>monospace</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>monospace italic</i>	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
monospace bold	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: <code>MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2></code>

Further reading

This section lists publications by ARM and by third parties.

See also:

- <http://infocenter.arm.com> for access to ARM documentation.
- <http://www.arm.com> for current errata, addenda, and Frequently Asked Questions.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *RealView Debugger Essentials Guide* (ARM DUI 0181)
- *RealView Debugger Target Configuration Guide* (ARM DUI 0182)
- *RealView Debugger Trace User Guide* (ARM DUI 0322)
- *RealView Debugger RTOS Guide* (ARM DUI 0323)
- *RealView Debugger Command Line Reference Guide* (ARM DUI 0175).

For details on using the compilation tools, see the books in the ARM Compiler toolchain documentation.

For details on using RealView Instruction Set Simulator, see the following documentation:

- *RealView Instruction Set Simulator User Guide* (ARM DUI 0207).

For details on using and configuring *Real-Time System Models* (RTSMs), see:

- *RealView Development Suite Real-Time System Model User Guide* (ARM DUI 0424).

For general information on software interfaces and standards supported by ARM tools, see:

install_directory\Documentation\Specifications\...

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

For details on ARM architectures, see:

- *ARMv6-M Architecture Reference Manual* (ARM DDI 0419)
- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).

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

- *DSTREAM Setting Up the Hardware* (ARM DUI 0481)
- *DSTREAM System and Interface Design Reference* (ARM DUI 0499)

- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities* (ARM DUI 0498)
- *RealView ICE and RealView Trace Setting Up the Hardware* (ARM DUI 0515)
- *RealView ICE and RealView Trace System and Interface Design Reference* (ARM DUI 0517).

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 a detailed introduction to regular expressions, as used in the RealView Debugger search and pattern matching tools, see:

Jeffrey E. F. Friedl, *Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools*, 1997. O'Reilly & Associates, Inc. ISBN 1-56592-257-3.

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.

For more information about the JTAG standard, see:

IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Std. 1149.1), available from the IEEE (<http://www.ieee.org>).

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any problems with this product, submit a Software Problem Report:

1. Select **Send a Problem Report...** from the RealView Debugger **Help** 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. E-mail the report to your supplier.

Feedback on this book

If you have comments on content then send an e-mail to errata@arm.com. Give:

- the title
- the number, ARM DUI 0153L
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1

RealView Debugger Features

This chapter describes the various RealView® Debugger features that are available for debugging single and multiprocessor targets. It contains the following sections:

- *Overview of RealView Debugger windows and views* on page 1-2
- *Target connection* on page 1-23
- *Image and binary loading* on page 1-26
- *TrustZone technology support* on page 1-28
- *Multiprocessor debugging* on page 1-29
- *Execution control* on page 1-31
- *Memory mapping* on page 1-32
- *Execution context and scope* on page 1-34
- *Breakpoints in RealView Debugger* on page 1-36
- *Examining the target execution environment* on page 1-38
- *Altering the target execution environment* on page 1-40
- *Command scripts* on page 1-42
- *Macros* on page 1-43
- *Log and journal files* on page 1-44
- *Editing source files* on page 1-45
- *Searching source files* on page 1-46.

1.1 Overview of RealView Debugger windows and views

RealView Debugger has many windows and views that enable you to:

- prepare your target for debugging
- monitor your target execution environment
- alter your target execution environment.

See also:

- *Common features of views* on page 1-3
- *Persistence of settings in specific views* on page 1-3
- *Toolbar buttons* on page 1-3
- Windows:
 - *Code window* on page 1-4
 - *Connect to Target window* on page 1-5
 - *Synchronization Control window* on page 1-7
 - *Analysis window* on page 1-8.
- Views:
 - *Home Page* on page 1-9
 - *Disassembly view* on page 1-9
 - *Source code view* on page 1-9
 - *Break/Tracepoints view* on page 1-10
 - *Call Stack view* on page 1-10
 - *Classes view* on page 1-12
 - *Comms Channel view* on page 1-13
 - *Diagnostic Log view* on page 1-14
 - *Locals view* on page 1-15
 - *Memory view* on page 1-16
 - *Output view* on page 1-17
 - *Process Control view* on page 1-18
 - *Registers view* on page 1-19
 - *Resource Viewer* on page 1-20
 - *Stack view* on page 1-20
 - *Symbols view* on page 1-21
 - *Watch view* on page 1-22
- the *Windows and Views* topic in the RealView Debugger Help for more details on the GUI components of the windows and views, and a summary of the features.

1.1.1 Common features of views

All views have the following features in common:

- You can hide and show views to customize your debug view as required.
- You can dock and float views for a Code window. However, if you have multiple Code windows displayed, you can only dock a view into the Code window that you used to display the view (that is, the parent Code window).
- A status bar, which identifies the connection in the parent Code window.
- A default name that reflects the view contents. However, you can rename a view if required. A view name can have a maximum of 64 characters.

1.1.2 Persistence of settings in specific views

Table 1-1 shows the settings that persist for specific views if they are visible when you exit RealView Debugger. The settings are not applied to new instances of the view. If you close a view and then re-open it, the settings return to the default.

Table 1-1 Settings that persist for specific views

View	Settings that persist
Memory	The Memory view toolbar settings.
Registers	The User tab.
Symbols	The filter.
Watch	Any watched variables.

1.1.3 Toolbar buttons

By default, toolbar buttons are displayed as small icons and without button names displayed. When referring to toolbar buttons, the RealView Debugger documentation uses the button names. You can independently change the display of the toolbar buttons for the Code window and the Analysis window.

Displaying the toolbar buttons as large icons

To display the toolbar buttons as large icons:

1. Right-click on a toolbar to display the context menu.
2. Select **Show Big Icons** from the context menu.

Displaying the button names

To see the button names:

1. Right-click on a toolbar to display the context menu.
2. Select **Show Toolbar Text** from the context menu.

See also

- *Code window* on page 1-4
- *Analysis window* on page 1-8.

1.1.4 Code window

This is the main window of RealView Debugger. All RealView Debugger operations are accessible from the main Code window. Figure 1-1 shows an example:

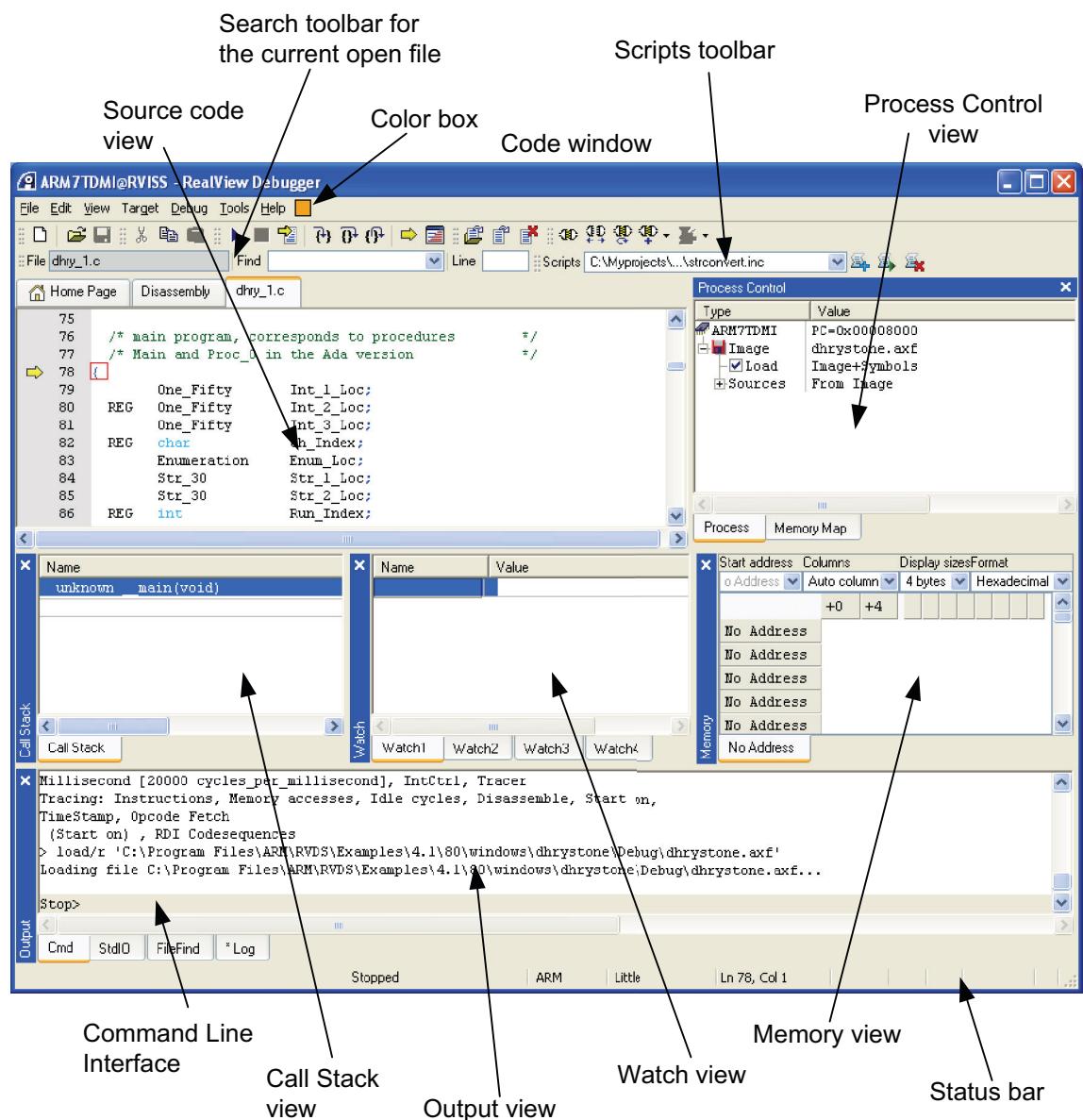


Figure 1-1 Code window (default layout)

Although a single Code window is sufficient for most tasks, you might want to open more Code windows if you are:

- debugging multiprocessor systems, where you can closely associate each Code window with a specific target connection
- debugging OS-aware connections, where you can closely associate each Code window with a specific thread.

Code window status bar

The Code window status bar shows:

- The run state of the target for the current connection, which can be:
 - Running
 - Stopped
 - nn Steps Remaining, when performing multi-step operations with CLI commands
 - Waiting, when performing synchronized stepping.
- The current processor state:
 - ARM
 - Jazelle
 - Thumb
 - Thumb2-EE.
- The target endianness:
 - Little
 - Big.
- The line number and column number of the cursor position in the **Disassembly** tab or a source code tab.
- A Script indicator to show that a script is running.
- When any log, journal, and STDIO log files are open:
 - LOG
 - JOU
 - STDIOLog.

See also

- *Multiprocessor debugging* on page 1-29
- *Command scripts* on page 1-42
- *Log and journal files* on page 1-44
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 2 *RealView Debugger Commands* for details of the LOG, JOURNAL, STDIOLOG, and the various step commands.
- *RealView Debugger RTOS Guide*.

1.1.5 Connect to Target window

The Connect to Target window enables you to:

- add Debug Configurations to a Debug Interface
- customize your Debug Configurations.
- connect to targets.

Connection groupings

There are two connection groupings available in the Connect to Target window:

- A Target grouping, which lists the individual targets available for each Debug Interface. The Target grouping corresponds to the Debug Interface that is used to access the debug target, for example RealView ICE.

Figure 1-2 shows an example:

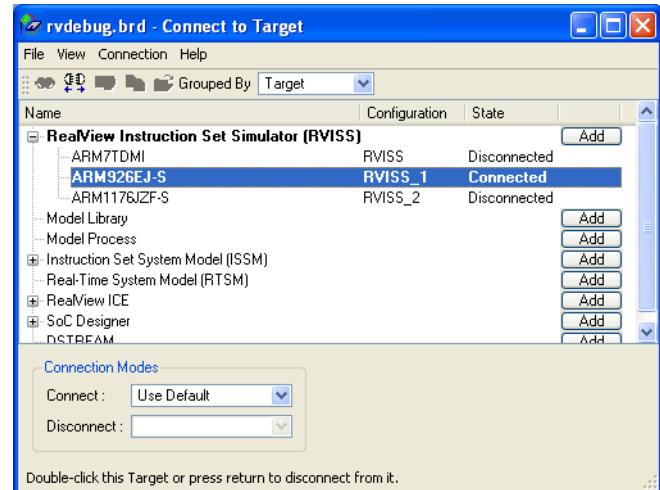


Figure 1-2 Connect to Target window (Target grouping)

- A Configuration grouping, which lists the targets for each Debug Configuration. Each Debug Configuration is listed under the related Debug Interface. RealView Debugger gives each Debug Configuration a default name, which you can change to a more meaningful name if required. Figure 1-3 shows an example:

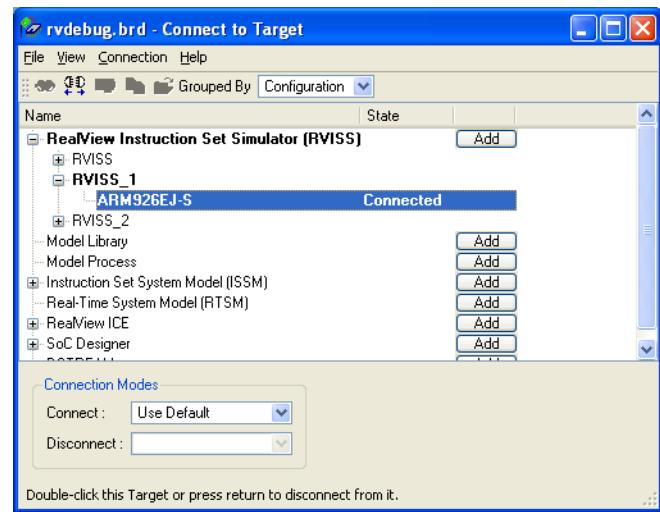


Figure 1-3 Connect to Target window (Configuration grouping)

Note

A *RealView ARMulator® ISS* (RVISS) Debug Configuration (RVISS in this example) can contain only a single processor. However, you can create multiple RVISS Debug Configurations, each with a different processor type.

Connections to trace components

If your development platform contains trace components, then you must add the trace components when you configure the Debug Interface targets.

If your development platform comprises a single processor with an *Embedded Trace Macrocell™* (ETM™) and, optionally, an *Embedded Trace Buffer™* (ETB™), then you do not have to connect to the trace components. For development platforms with more complex trace components, you must connect to and configure all trace components if you want to perform tracing. When the current connection is to a trace component, then for that connection:

- the Code window supports only the Registers view, and Memory view where appropriate
- all execution-related debugging operations are disabled.

See also

- *About creating a Debug Configuration* on page 3-8
- *Chapter 3 Target Connection*
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Chapter 3 Customizing a Debug Configuration*
- the following in the *RealView Debugger Trace User Guide*:
 - *Chapter 4 Configuring the ETM*.
- *DSTREAM Setting Up the Hardware* (ARM DUI 0481)
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities* (ARM DUI 0498)
- *RealView ICE and RealView Trace Setting Up the Hardware* (ARM DUI 0515).

1.1.6 Synchronization Control window

The Synchronization Control window enables you to set up synchronized actions, synchronized execution operations, and cross-triggering when debugging multiprocessor applications. Figure 1-4 shows an example:

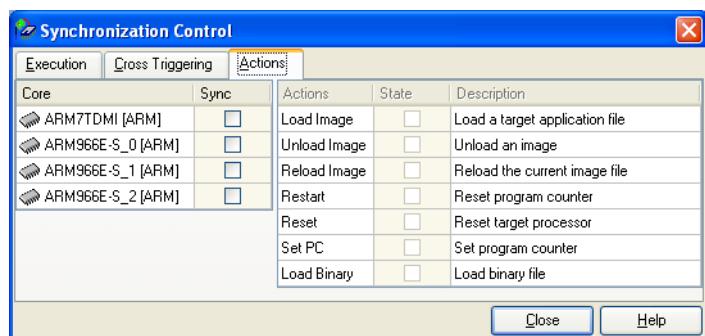


Figure 1-4 Synchronization Control window

Although you can set up synchronization and cross-triggering for multiple RVISS targets, they cannot share memory. To debug a true multiprocessor system, you must use either:

- a hardware multiprocessor system with a DSTREAM or RealView ICE unit
- a simulated multiprocessor system using:
 - *Instruction Set System Model* (ISSM)

- Model Library
- Model Process
- *Real-Time System Model (RTSM)*.
- Carbon SoC Designer Plus.

See also

- *Multiprocessor debugging* on page 1-29.

1.1.7 Analysis window

The Analysis window enables you to configure trace and analyze trace output passed to RealView Debugger. The trace output can be captured:

- directly from an ETM using a RealView ICE unit in conjunction with a RealView Trace or RealView Trace 2 unit.
- from an ETB using a DSTREAM or RealView ICE unit.

Note

RealView Debugger does not support tracing from the external trace port of a SoC with DSTREAM.

Figure 1-5 shows an example of the Analysis window.

Elem	Time/cycl	Type	Symbolic	Address	Opcode	Other
*0	6	Instr	_scatterload_rt2	0x00008008	0xE28F002C	ADR
1	7	Instr	_scatterload_rt2	0x0000800C	0xE8900C00	LDM
2	11	Instr	_scatterload_rt2	0x00008010	0xE08AA000	ADD
3	12	Instr	_scatterload_rt2	0x00008014	0xE0BB0000	ADD
4	13	Instr	_scatterload_rt2	0x00008018	0xE24A7001	SUB
5	14	Instr	_scatterload_null	0x0000801C	0xE15A000B	CMP
6	15	Instr	_scatterload_null	0x00008020	0x1A000000	BNE
7	18	Instr	_scatterload_null	0x00008028	0xE8BA000F	LDM
8	24	Instr	_scatterload_null	0x0000802C	0xE24FE018	ADR
9	25	Instr	_scatterload_null	0x00008030	0xE3130001	TST
10	26	Instr	_scatterload_null	0x00008034	0x1047F003	SUBNE
11	27	Instr	_scatterload_null	0x00008038	0xE1A0F003	MOV
12	30	Instr	_scatterload_zeroinit	0x00008044	0xE3B03000	MOVS
13	31	Instr	_scatterload_zeroinit	0x00008048	0xE3B04000	MOVS
14	32	Instr	_scatterload_zeroinit	0x0000804C	0xE3B05000	MOVS
15	33	Instr	_scatterload_zeroinit	0x00008050	0xE3B06000	MOVS
16	34	Instr	_scatterload_zeroinit	0x00008054	0xE2522010	SUBS
17	35	Instr	_scatterload_zeroinit	0x00008058	0x28A10078	STMCS

Figure 1-5 Analysis window

See also

- the following in the *RealView Debugger Trace User Guide*:
 - Chapter 9 *Analyzing Trace with the Analysis Window*
 - Appendix A *Setting up the Trace Hardware*
 - Appendix B *Setting up the Trace Software*.

1.1.8 Home Page

The **Home Page** tab, shown in Figure 1-1 on page 1-4, keeps track of the last 10 distinct target connections you established and the last 10 distinct images that you loaded. Each connection and image entry is displayed as a blue hyperlink. To connect to a target or load an image, click the appropriate hyperlink.

When a target is connected:

- the hyperlink for that target is disabled
- the text (connected) is displayed after the connection name.

Figure 1-6 shows an example **Home Page**, with a connection established an ARM926EJ-S™ processor in the RVISS_1 Debug Configuration:

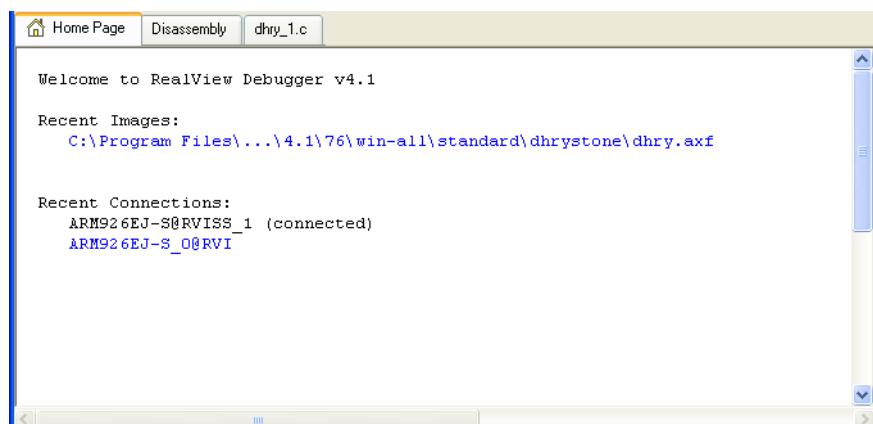


Figure 1-6 Home Page

See also

- *Making a connection from the Home Page* on page 3-28
- *Loading an image from the Home Page* on page 4-6.

1.1.9 Disassembly view

The disassembly view is available in the **Disassembly** tab, shown in Figure 1-6, and enables you to:

- view the disassembly of your application
- navigate to various points in the disassembly
- perform tasks such as setting breakpoints and changing the execution context.

See also

- *Execution context and scope* on page 1-34
- *Breakpoints in RealView Debugger* on page 1-36
- *Chapter 5 Navigating the Source and Disassembly Views*.

1.1.10 Source code view

The source code view enables you to:

- view sources for loaded images
- edit source files
- navigate to various points in your source files

- perform tasks such as setting breakpoints and changing the execution context.

Figure 1-1 on page 1-4 shows an example.

Source files are opened as read-only automatically by RealView Debugger when execution stops at a line of source in that file. A prompt is displayed if you attempt to edit a source file opened in this way. However, if you open a source file manually, then you can edit it directly.

See also

- *Execution context and scope* on page 1-34
- *Breakpoints in RealView Debugger* on page 1-36
- *Editing source files* on page 1-45
- Chapter 5 *Navigating the Source and Disassembly Views*.

1.1.11 Break/Tracepoints view

The Break/Tracepoints view enables you to:

- view and edit breakpoints
- view and edit tracepoints.

Figure 1-7 shows an example:

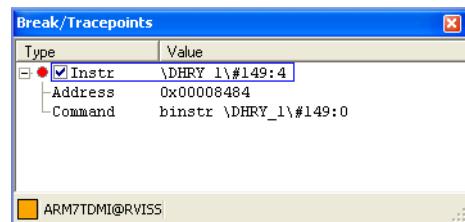


Figure 1-7 Break/Tracepoints view

See also

- *Breakpoints in RealView Debugger* on page 1-36
- the following in the *RealView Debugger Trace User Guide*:
 - Chapter 5 *Tracepoints in RealView Debugger*.

1.1.12 Call Stack view

The Call Stack view enables you to:

- monitor the Call Stack during target execution
- set a breakpoint on an entry in the Call Stack
- move the context up and down the Call Stack as required.

Figure 1-8 on page 1-11 shows an example:

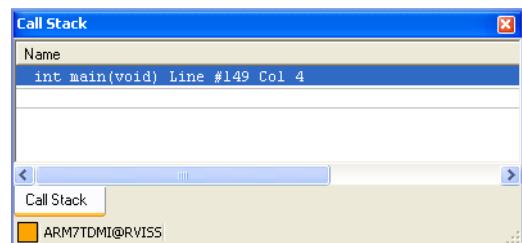


Figure 1-8 Call Stack view

The Call Stack view is visible and docked into the Code window by default (see Figure 1-1 on page 1-4).

See also

- *Examining the target execution environment* on page 1-38.

1.1.13 Classes view

The Classes view enables you to:

- view C++ classes in applications that are built with C++ source files
- set breakpoints on the member functions of a class.

Figure 1-9 shows an example:

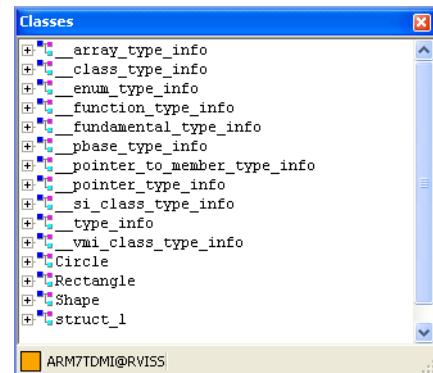


Figure 1-9 Classes view

See also

- *Viewing C++ classes* on page 13-22.

1.1.14 Comms Channel view

The EmbeddedICE® logic in ARM processors contains a *Debug Communications Channel* (DCC). This enables data to be passed between the target and the RealView Debugger using the JTAG port and a protocol converter without stopping the program flow or entering debug state. The Comms Channel view enables you to communicate with the target over DCC.

Figure 1-10 shows an example:

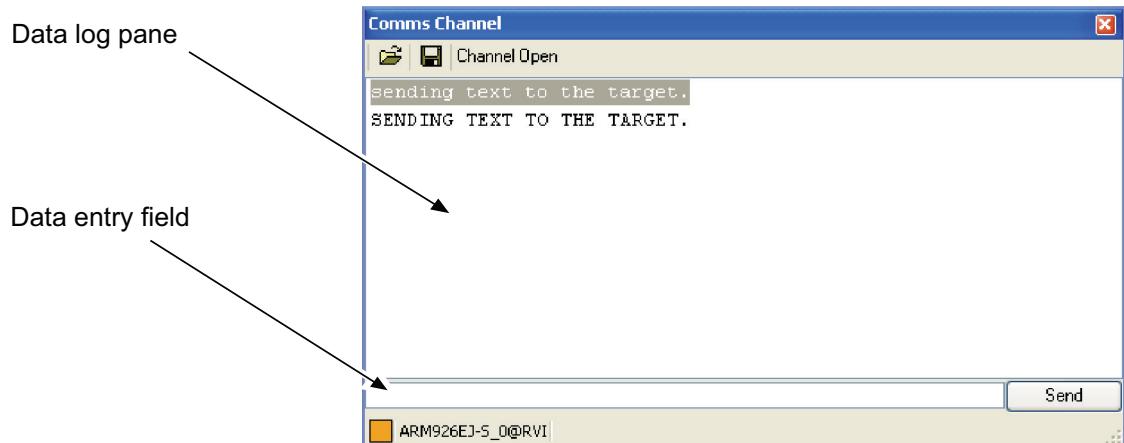


Figure 1-10 Comms Channel view

By default, data sent to the target is displayed in the Data log pane, and appears as grey text. You can choose to hide the data sent to the target from the Data log pane.

Data received from the target is displayed as black text in the Data log pane.

See also

- *Altering the target execution environment* on page 1-40.

1.1.15 Diagnostic Log view

The Diagnostic Log view enables you to:

- view the messages generated during target connection
- filter out unwanted connection messages.

Figure 1-11 shows an example:

The screenshot shows a Windows-style application window titled "Diagnostic Log". The window has a toolbar with icons for search, refresh, and exit. A status bar at the bottom displays the text "ARM926EJ-S_0@RVI". The main area is a table with columns: #, Time, Component, Severity, Device Name, and Message. There are 6 rows of data:

#	Time	Component	Severity	Device Name	Message
0	10:03:20.227090	Registers	Info	ARM926EJ-S_0	The following register numbers are Range 0x0 to 0x2c Range 0x100 to 0x116 Range 0x200 to 0x27f Mask 0x800, value 0x800 Mask 0xfc00, value 0x7c00
1	10:03:20.242715	Breakpoints	Info	ARM926EJ-S_0	2 hardware breakpoints supported.
2	10:03:20.242715	Breakpoints	Info	ARM926EJ-S_0	Processor Exceptions are available
3	10:03:20.258340	Target Connection	Info	ARM926EJ-S_0	DCC channel is available
4	10:03:20.258340	Target Connection	Info	ARM926EJ-S_0	RealViewICE supports tightly-coupled memory access
5	10:03:20.289591	Target Connection	Info	ARM926EJ-S_0	Searching for core properties in 'C:\Program Files\ARM\RealView\ARM926EJ-S\ARM926EJ-S_0\ARM926EJ-S_0.dcc'
6	10:03:20.305216	Target Connection	Info	ARM926EJ-S_0	No match for core type ARM926EJ-S_0

Figure 1-11 Diagnostic Log view

See also

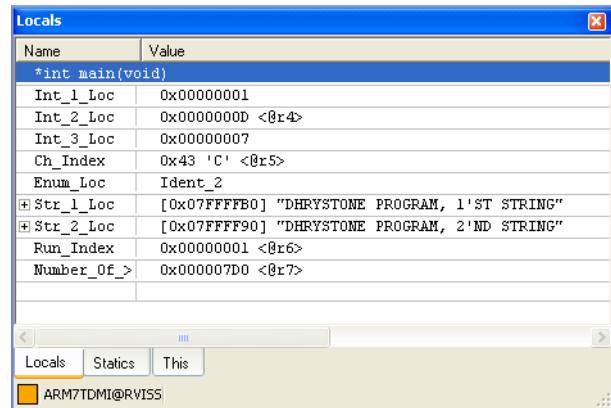
- *Examining details in the Diagnostic Log view* on page 3-63.

1.1.16 Locals view

The Locals view enables you to perform the following operations on local variables, static symbols, and this objects:

- view the contents
- set a breakpoint
- view memory at the address
- view memory at the value
- change the value.

Figure 1-12 shows an example:



The screenshot shows a Windows-style dialog titled "Locals". It contains a table with two columns: "Name" and "Value". The table lists several variables and their current values. Some values are memory addresses, while others are strings. The bottom of the window has tabs for "Locals", "Statics", and "This", with "Locals" being the active tab. A status bar at the bottom right shows the text "ARM7TDMI@RVISS".

Name	Value
*int main(void)	
Int_1_Loc	0x00000001
Int_2_Loc	0x0000000D <0r4>
Int_3_Loc	0x00000007
Ch_Index	0x43 'C' <0r5>
Enum_Loc	Ident_2
+ Str_1_Loc	[0x07FFFFB0] "DHRYSTONE PROGRAM, 1'ST STRING"
+ Str_2_Loc	[0x07FFFF90] "DHRYSTONE PROGRAM, 2'ND STRING"
Run_Index	0x00000001 <0r6>
Number_0f_>	0x000007D0 <0r7>

Figure 1-12 Locals view

See also

- *Examining the target execution environment* on page 1-38
- *Altering the target execution environment* on page 1-40.

1.1.17 Memory view

The Memory view enables you to:

- display multiple areas of target memory in different tabbed views
 - display the ASCII equivalent of the memory contents
 - display the memory contents in different formats
 - display the memory locations in different data sizes
 - display *Memory Management Unit* (MMU) page tables, where supported
 - change the values of specific memory locations
 - set breakpoints at selected memory locations
 - drag and drop a tabbed view from one Memory view to another.

Figure 1-13 shows an example:

Start address	Columns		Data sizes		Format	
0x00000000	Auto column		Default		Hexadecimal	
	+0	+1	+2	+3	+4	+5
0x00000000	0x00	0x00	0x00	0xEB	0xB0	0x07
0x00000006	0x00	0xEB	0x2C	0x00	0x8F	0xE2
0x0000000C	0x00	0x0C	0x90	0xE8	0x00	0xA0
0x00000012	0x8A	0xE0	0x00	0xB0	0x8B	0xE0
0x00000018	0x01	0x70	0x4A	0xE2	0x0B	0x00
0x0000001E	0x5A	0xE1	0x00	0x00	0x00	0x1A
0x00000024	0xA8	0x07	0x00	0xEB	0x0F	0x00
0x0000002A	0xBA	0xE8	0x18	0xE0	0x4F	0xE2
0x00000000						0

Figure 1-13 Memory view

The Memory view is visible and docked into the Code window by default (see Figure 1-1 on page 1-4).

See also

- *Examining the target execution environment* on page 1-38
 - *Altering the target execution environment* on page 1-40.

1.1.18 Output view

The Output view enables you to:

- view messages and commands generated by RealView Debugger
- view messages and prompts generated during application execution
- view the results of multi-file search operations
- copy the displayed messages and commands
- enter CLI commands
- paste CLI commands at the command prompt.

Figure 1-14 shows an example:

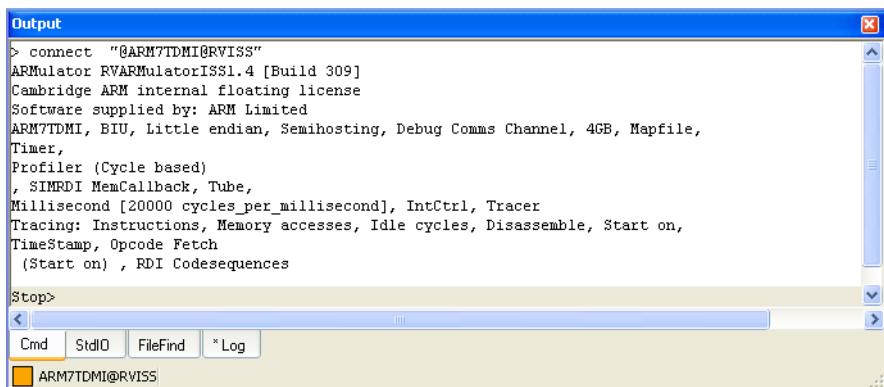


Figure 1-14 Output view

Note

You can also capture messages and commands generated by RealView Debugger or an application to log and journal files.

The Output view is visible and docked into the Code window by default (see Figure 1-1 on page 1-4). Only one instance of the Output view can be displayed for a Code window.

The Output view context menu includes the following options:

Format... Displays an Encoding dialog box where you can select the one of the following encodings:

- ASCII (the default)
- UTF-8
- Locale.

Clear Clear all displayed text.

See also

- *Log and journal files* on page 1-44
- *Searching source files* on page 1-46
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 2 *RealView Debugger Commands*.

1.1.19 Process Control view

The Process Control view enables you to:

- view process details
- load, unload, and reload images
- map target memory
- view thread details when debugging applications on an OS-aware connection.

Figure 1-15 shows an example:

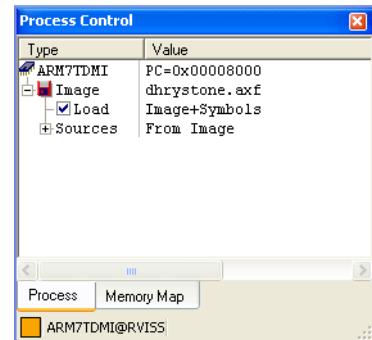


Figure 1-15 Process Control view

The Process Control view is visible and docked into the Code window by default (see Figure 1-1 on page 1-4).

See also

- *Image and binary loading* on page 1-26
- *Memory mapping* on page 1-32
- the following in the *RealView Debugger RTOS Guide*:
 - *Examining thread details in the Thread tab* on page 5-8.

1.1.20 Registers view

The Registers view enables you to:

- monitor the registers during target execution
- change the values of registers
- view the memory at the value of register
- create your own custom register view.

Figure 1-16 shows an example:

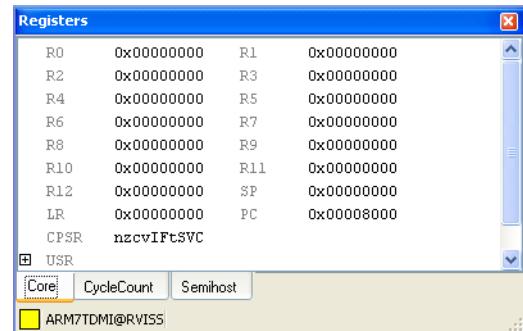


Figure 1-16 Registers view

Depending on the target associated with the current connection and the debug interface you are using, additional tabs might be visible in the Registers view, such as:

- a **CP15** tab that displays and sets the values of registers in coprocessor 15 (the System Control coprocessor)
- a **Cache Operations** tab that you can use to perform operations on the cache for the target
- a **TLB Operations** tab that you can use to perform operations on the *translation look-aside buffer* (TLB) for the target
- if you connect to a target through DSTREAM or RealView ICE, a **Debug** tab that controls various internal debugger settings, many of which are specific to debug interface hardware.

See also

- *Examining the target execution environment* on page 1-38
- *Altering the target execution environment* on page 1-40.

1.1.21 Resource Viewer

The Resource Viewer enables you to view resources on target connections. However, the Resource Viewer is especially useful when debugging OS-aware targets. Figure 1-17 shows an example:

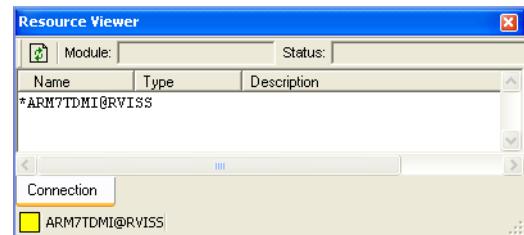


Figure 1-17 Resource Viewer

See also

- the following in the *RealView Debugger RTOS Guide*:
 - Chapter 6 *Viewing OS Resources*.

1.1.22 Stack view

The Stack view, enables you to:

- monitor the Stack during target execution
- alter a value on the Stack
- display the ASCII equivalent of the Stack contents
- display the symbol at a Stack address, or at the value in the Stack address
- set a breakpoint or tracepoint on a Stack address.

Figure 1-18 shows an example:

Stack		
0x00000000SP	0xE7FF0010	
0x00000004	0xE800E800	
0x00000008	0xE7FF0010	
0x0000000C	0xE800E800	
0x00000010	0xE7FF0010	
0x00000014	0xE800E800	
0x00000018	0xE7FF0010	
0x0000001C	0xE800E800	
0x00000020	0xE7FF0010	
0x00000024	0xE800E800	
0x00000028	0xE7FF0010	
0x0000002C	0xE800E800	

Figure 1-18 Stack view

See also

- *Examining the target execution environment* on page 1-38
- *Altering the target execution environment* on page 1-40.

1.1.23 Symbols view

The Symbols view enables you to:

- view and locate modules, functions, and variables in loaded images
- filter the names of modules, functions, and variables
- view the source or disassembly of a function
- set a breakpoint on a function or variable.

Figure 1-19 shows an example:



Figure 1-19 Symbols view

See also

- *Examining the target execution environment* on page 1-38.

1.1.24 Watch view

The Watch view enables you to:

- view and set watch variables
- set a breakpoint at the value of a watched variable
- view memory at the address of a watched variable
- view memory at the value of a watched variable
- change the values of watch variables.

Figure 1-20 shows an example:

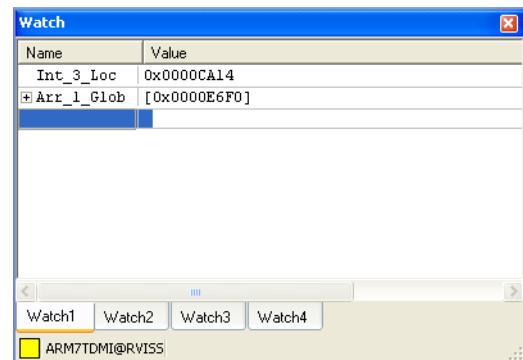


Figure 1-20 Watch view

The Watch view is visible and docked into the Code window by default (see Figure 1-1 on page 1-4).

See also

- *Examining the target execution environment* on page 1-38
- *Altering the target execution environment* on page 1-40.

1.2 Target connection

Target connection is the mechanism that RealView Debugger uses to access your debug target. Targets, such as ARM® architecture-based processors, are accessed through a debug target interface. The debug target interface is referred to in RealView Debugger as a Debug Interface, which can provide the interface to hardware or software development platforms:

- An ARM development board with one or more processors, accessed through DSTREAM or RealView ICE, is an example of a hardware development platform.
- RVISS, *Instruction Set System Model* (ISSM), and *Real-Time System Model* (RTSM) are examples of software development platforms, and are provided with *RealView Development Suite* (RVDS). RealView Debugger also supports the following simulated development platforms:
 - Model Library
 - Model Process
 - Carbon SoC Designer Plus.

Before you can connect to a target processor, you might have to configure the appropriate Debug Interface and connection details.

See also:

- *Debug Interface configuration overview*
- *Debug Configuration overview* on page 1-24
- *Default Debug Configurations provided with RVDS* on page 1-24
- *Files used to describe target connections and configurations* on page 1-25
- Chapter 3 *Target Connection* for details on how to connect to target processors.

1.2.1 Debug Interface configuration overview

You must configure a Debug Interface to recognize the targets on the development platform to which it is attached. You must create a Debug Configuration before you can connect to the targets on your development platform. However, default Debug Configurations are provided for some simulated targets.

See also

- *Default Debug Configurations provided with RVDS* on page 1-24
- *About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 2 *Customizing a Debug Interface configuration*.

1.2.2 Debug Configuration overview

A Debug Configuration enables you to customize the debugging environment for your development platform, such as the memory map. It describes how RealView Debugger connects to, and interacts with, your development platform.

A Debug Configuration:

- References a Debug Interface configuration file that identifies the targets on your development platform. Each Debug Interface has a specific configuration dialog box that you use to modify the related configuration file.
- References other configuration files that describe memory map related details for the individual components of your development platform, such as the development board and the target processor. These configuration files are called *Board/Chip Definition* (BCD) files. You can view, modify, and create BCD files using the Connection Properties window.
- Directly defines other debugging features, such as OS-awareness, that enable you to customize the debugging environment to your specific requirements.

You add, rename, and delete a Debug Configuration using the Connect to Target window. To customize a Debug Configuration then use:

- The Connection Properties dialog box for the more commonly used settings in a Debug Configuration. The steps required to customize the settings on this dialog box are described in this book.
- The Connection Properties window for more advanced customizations. These customizations are fully described in the *RealView Debugger Target Configuration Guide*.

See also

- *Default Debug Configurations provided with RVDS*
- *About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 2 *Customizing a Debug Interface configuration*
 - Chapter 3 *Customizing a Debug Configuration*
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

1.2.3 Default Debug Configurations provided with RVDS

Table 1-2 shows the default Debug Configurations provided with RVDS:

Table 1-2 Default Debug Configurations

Debug Configuration	Simulated target
RVISS	ARM7TDMI®
RVISS_1	ARM926EJ-S
RVISS_2	ARM1176JZF-S™
ISSM	Cortex™-A8

If these default configurations meet your target debug requirements, you can connect to the simulated targets without having to configure them.

1.2.4 Files used to describe target connections and configurations

RealView Debugger uses a *board file* to access information about the debugging environment and the targets available on your development platform. The board file contains an entry for each Debug Configuration that you create.

Default board file

When you start RealView Debugger for the first time after installation, RealView Debugger:

- Creates a user-specific RealView Debugger home directory for you in:
`C:\Documents and Settings\username\Application Data\ARM\rvdebug\version`
- Copies the default rvdebug.brd file to your user-specific RealView Debugger home directory. Any subsequent changes you make to the connection configuration are then stored in this user-specific rvdebug.brd file.

User-specific board file

The user-specific rvdebug.brd file includes connection configuration settings and references to other files that are required to define a Debug Configuration, such as:

- Debug Interface configuration files that identify the targets to which you can connect
- BCD files that specify memory map related details for various components of the target hardware.

Board/Chip Definition files

When you start RealView Debugger for the first time after installation, RealView Debugger copies the installed BCD files, and other configuration files, to your default settings directory:

`C:\Documents and Settings\userID\Local Settings\Application
Data\ARM\rvdebug\version\shadowbase\etc`

Having separate BCD files for specific components of your development platform means that you can re-use them for:

- other Debug Configurations that specify different debugging environments for your development platform
- other development platforms that have the same components.

For example, if you have an ARM966E-S™ processor module on an Integrator™/CP board, then you can use the CM966ES.bcd and CP.bcd files. Because the board definitions are separate from the processor definitions, if you change the processor module on the Integrator/CP to an ARM920T™, then you have only to change the reference to the processor module BCD file CM920T.bcd. Also, if you have multiple processor modules on the Integrator/CP, then you can reference the BCD file for each one.

See also

- *Redefining the RealView Debugger directories* on page 2-9
- *Files in the default settings directory* on page B-3
- *Files in the home directory* on page B-5.

1.3 Image and binary loading

After connecting to your debug target, you must load your code to the target before you can debug it. The application might include:

- one or more executable images
- one or more binary files
- an operating system kernel, such as Linux, together with shared libraries (that is, an OS-aware application).

This section describes the various mechanisms available to load the different types of files that make up your application.

See also:

- *Executable images*
- *Binaries*
- *Multiple images and binaries*
- *RealMonitor connections* on page 1-27
- *OS-aware connections* on page 1-27
- Chapter 4 *Loading Images and binaries*.

1.3.1 Executable images

For an executable image, you can:

- load the image, which by default loads the symbols and all sections
- load only the symbols
- specify the values for any arguments that your image uses
- load specific sections of the image
- unload and reload the image.

When you load an executable image, the image details are shown in the **Process** tab of the Process Control view (see Figure 1-15 on page 1-18). You can also perform the load, unload, and reload operations in the **Process** tab. In addition, if memory mapping is enabled, the memory map in the **Memory Map** tab is updated to show the related details for the image.

See also

- *Memory mapping* on page 1-32.

1.3.2 Binaries

Binaries are loaded in a different way to executable images. When you load a binary no details are shown in the **Process** tab of the Process Control view (see Figure 1-15 on page 1-18). However, the memory map might be updated to show the related details for the binary.

See also

- *Memory mapping* on page 1-32
- *Loading a binary* on page 4-12.

1.3.3 Multiple images and binaries

You can load multiple images and binaries to the same target. If you do this, you must make sure that each image or binary does not occupy the same area of memory, or does not overlap a memory area used by another image or binary.

See also

- *Loading multiple images to the same target* on page 4-14.

1.3.4 RealMonitor connections

RealMonitor enables you to perform nonstop debugging on an application in a real-time environment. To debug a RealMonitor-enabled application, you must create two RealView ICE Debug Configurations to the target:

- one connection to load and run the application
- one connection to control RealMonitor and perform the debugging.

See also

- *About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43.

1.3.5 OS-aware connections

Typically, an OS-aware application comprises an OS kernel, such as ARM Embedded Linux, and shared library files. The loading and debugging of OS-aware images requires that you create a custom Debug Configuration that sets up the required OS-aware debugging environment for your development platform.

See also

- *About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger RTOS Guide*:
 - Chapter 1 *OS Support in RealView Debugger*.

1.4 TrustZone technology support

For targets that support TrustZone® technology, the following sections describe the features that are supported in RealView Debugger.

See also:

- *Determining the current world*
- *Specifying addresses.*

1.4.1 Determining the current world

The current world for a target that supports TrustZone technology is shown in the current world indicator of the Code window status bar:



- Indicates that the target is in Secure World. This is the initial state after connection.
- Indicates that the target is in Normal World.

1.4.2 Specifying addresses

Addresses on a target that supports TrustZone technology are prefixed with:

S: For an address in Secure World memory.

N: For an address in Normal World memory.

If you want to specify an address in the current world, then you can omit the prefix.

Some example are:

- S:0x1000
- S:\DHRY_1\149:0
- S:Arr_1_Glob
- N:@PC.

See also

- *Memory mapping* on page 1-32
- *Breakpoints in RealView Debugger* on page 1-36
- *Examining the target execution environment* on page 1-38

1.5 Multiprocessor debugging

You can connect to multiple targets simultaneously and debug applications on each of those targets. This section describes the various features that enable you to perform multiprocessor debugging.

See also:

- *Multiple target connections*
- *Multiple Code windows*
- *Processor synchronization* on page 1-30
- *Cross-triggering* on page 1-30
- *Connecting to multiple targets* on page 3-46
- Chapter 7 *Debugging Multiprocessor Applications*.

1.5.1 Multiple target connections

You can make connections to multiple targets. These connections can be to targets that are accessible through the same Debug Interface, or that are accessible through multiple Debug Interfaces. Targets can be ARM architecture-based processors.

When you have multiple connections, the last connection becomes the *current connection*. By default, the Code window shows the details for the current connection only. However, you can change the current connection so that the details for another connection become visible in the Code window.

See also

- *Target connection* on page 1-23.

1.5.2 Multiple Code windows

By default, only one RealView Debugger Code window is open, but you can open more Code windows if required.

When you have multiple target connections, you might want to view the details for more than one connection at the same time. To do this, you must have a Code window open for each connection you want to view. You can closely associate each Code window with a different target connection. The mechanism to do this is called *window attachment*. When a Code window is attached to a connection, only the details for that connection are visible in that Code window, even if you change the current connection.

————— Note —————

When you exit RealView Debugger the setup of your Code windows and connections is stored in the workspace. Therefore, when you next start RealView Debugger, the previous setup is restored.

See also

- Chapter 17 *Configuring Workspace Settings*.

1.5.3 Processor synchronization

Processor synchronization allows:

- An action, such as an image load, to be performed for all synchronized processors.
- An execution operation performed on one processor to affect the operation of another processor. For example, when you step one processor, then another processor also steps.

RealView Debugger enables you to synchronize processors using the Synchronization Control window.

See also

- *Synchronization Control window* on page 1-7
- *Chapter 7 Debugging Multiprocessor Applications*.

1.5.4 Cross-triggering

Cross-triggering is a feature that allows one processor to stop another processor. For example, when a breakpoint activates and stops a processor, then the other processors that are participating in cross-triggering also stop.

Usually, when you use cross-triggering, you set the Out trigger for one processor, and the In trigger for the other processors. Therefore, if a breakpoint is activated on the processor with the Out trigger, then those processors with the In trigger set also stop. However, a breakpoint that activates on any processor with the In trigger does not stop any other processor.

RealView Debugger enables you to set up cross-triggering using the Synchronization Control window.

See also

- *Synchronization Control window* on page 1-7
- *Chapter 7 Debugging Multiprocessor Applications*.

1.6 Execution control

RealView Debugger enables you to control the execution of your target applications. You can:

- start and stop execution
- step by a line of source code or multiple lines of source code, either:
 - into function (high-level step into)
 - over functions (high-level step over)
- step by an assembler instruction or multiple assembler instructions, either:
 - into function (low-level step into)
 - over functions (low-level step over)
- step until a specific C-style condition is reached
- step until a function returns, if execution is currently stopped inside the function
- run until a specific point is reached, which can be one of the following:
 - the current cursor position in a source file
 - a chosen line of source or disassembly.

— Note —

You can also use breakpoints to control execution. A breakpoint can test when an instruction at a specific address is executed, or when data is accessed at a specific location.

See also:

- *Breakpoints in RealView Debugger* on page 1-36
- *Chapter 8 Executing Images*.

1.7 Memory mapping

RealView Debugger supports memory mapping which enables you to tailor the memory view to match that of your target. A memory map contains user-defined regions that have attributes such as read-write, read-only, or write-only. This enables you to only modify and view memory that is valid for your target.

By default, the whole memory range (0x00000000 to 0xFFFFFFFF) is available to you in RealView Debugger, through either:

- the disassembly view, which is provided in the **Disassembly** tab of the Code window
- the Memory view.

See also:

- *Specifying a memory map*
- *Viewing the memory map*
- *Memory maps for targets that support TrustZone technology* on page 1-33.

1.7.1 Specifying a memory map

Memory mapping is required to make sure that RealView Debugger treats certain areas of memory correctly. For example, to program a Flash device on your development platform you must define a Flash memory area in your memory map. This Flash memory area must also reference the algorithms required to program your Flash device.

You can specify a memory map in the following ways:

- by creating a temporary memory map that exists for as long as the connection to the target exists
- by configuring the memory map of each target and the development board in separate BCD files.

See also

- Chapter 6 *Writing Binaries to Flash*
- Chapter 9 *Mapping Target Memory*.

1.7.2 Viewing the memory map

The memory map for your development platform is displayed in the **Memory Map** tab of the Process Control view (see Figure 1-15 on page 1-18). By default, memory mapping is disabled for an ARM architecture-based target connection.

When you connect to a target, the memory map can include the target-specific memory map and the development board-specific memory map. If your development platform has multiple targets, then the memory map for each target connection might show the same memory areas specific to your development board.

See also

- *Memory view* on page 1-16
- *Process Control view* on page 1-18
- Chapter 9 *Mapping Target Memory*.

1.7.3 Memory maps for targets that support TrustZone technology

For a target that supports TrustZone technology, the **Memory Map** tab shows the memory map for both the Normal World and Secure World.

See also

- *Memory view* on page 1-16
- *Process Control view* on page 1-18
- *TrustZone technology support* on page 1-28
- Chapter 9 *Mapping Target Memory*.

1.8 Execution context and scope

The execution context determines the visibility of variables and functions. A variable or function is referred to as *in scope* if the name can be accessed at the current point of execution. The scope of a variable or function can be:

- the current source file, for global variables and functions
- the current function, for local variables.

When you load an image, scope is initially set to the value of the PC, which is usually the entry point of the image. You can:

- change the scope to another part of your image
- determine the current location of the scope, that is, the execution context.

If the scope is at a location that corresponds to a source file, then RealView Debugger automatically opens that source file if the **Home Page** tab has the focus.

See also:

- *Displaying the location of the PC in the disassembly view*
- *Displaying the location of the PC in a source file* on page 1-35.

1.8.1 Displaying the location of the PC in the disassembly view

 To display the location of the PC in the disassembly view, click the **Locate PC** button on the Debug toolbar. The instruction corresponding to the address in the PC is displayed, and a yellow right arrow indicates the location of the PC. Figure 1-21 shows an example:

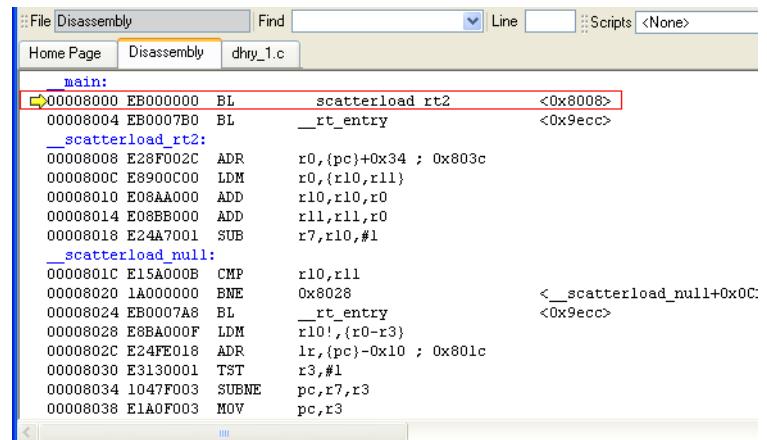


Figure 1-21 Location of PC in disassembly view

See also

- Chapter 10 *Changing the Execution Context*.

1.8.2 Displaying the location of the PC in a source file

To display the location of the PC in a source file:

1. Click the **Home Page** tab in the Code window.
2. Click the **Locate PC** button on the Debug toolbar. The source file containing the line of source associated with the PC is displayed. A yellow right arrow and red box indicates the location of the PC. Figure 1-22 shows an example:

```

File dhry_1.c Find Line Scripts <None>
Home Page Disassembly dhry_1.c
75
76     /* main program, corresponds to procedures */
77     /* Main and Proc_0 in the Ada version */
78     ( )
79     One_Fifty      Int_1_Loc;
80     REG   One_Fifty      Int_2_Loc;
81     One_Fifty      Int_3_Loc;
82     REG   char          Ch_Index;
83     Enumeration    Enum_Loc;
84     Str_30         Str_1_Loc;
85     Str_30         Str_2_Loc;
86     REG   int          Run_Index;
87     REG   int          Number_Of_Runs;
88
89     /* Initializations */
90
91     Next_Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
92     Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));

```

Figure 1-22 Location of the PC in a source file

See also

- Chapter 10 *Changing the Execution Context*.

1.9 Breakpoints in RealView Debugger

Breakpoints enable you to control target execution when certain events occur, and when certain conditions are met:

- Events determine when a breakpoint is hit.
- By default, when a breakpoint is hit, it is immediately activated and execution stops.
- By assigning conditions to the breakpoint, you can control when a breakpoint gets activated.

See also:

- *Events that determine when a breakpoint is hit*
- *Actions that can be performed when a breakpoint activates*
- *Conditional breakpoint activation* on page 1-37.

1.9.1 Events that determine when a breakpoint is hit

You can set breakpoints that are hit when the following events occur:

- the PC matches the address of the breakpoint
- a memory location is accessed in a particular way (read-only, write-only, or read and write)
- a specific data value is accessed at a specific memory location (read-only, write-only, or read and write)
- a specific data value is accessed at any memory location (read-only, write-only, or read and write).

See also

- Chapter 11 *Setting Breakpoints*.

1.9.2 Actions that can be performed when a breakpoint activates

By default, when a breakpoint activates, execution stops. You can also set a breakpoint that performs one or more of the following actions:

- continue execution
- output messages to the Code window, or to a user-defined window or file
- update one or more windows and views.

See also

- Chapter 12 *Controlling the Behavior of Breakpoints*.

1.9.3 Conditional breakpoint activation

You can assign one or more of the following conditions to a breakpoint to delay activation of that breakpoint:

- after the breakpoint is hit a specified number of times
- when the result of an expression is either true or false
- when a specific instance of a C++ object is executed.

These are software conditions, because they are controlled by RealView Debugger. Although you can assign conditions to hardware breakpoints that are controlled by hardware, it is the software conditions that RealView Debugger uses to indicate that a breakpoint is conditional.

See also

- *Unconditional and conditional breakpoints* on page 11-6
- Chapter 12 *Controlling the Behavior of Breakpoints*.

1.10 Examining the target execution environment

RealView Debugger provides many features to enable you to examine various items on your target when it stops at specific points.

See also:

- *Debug views*
- *Other methods of examining the target execution environment.*

1.10.1 Debug views

The following debug views are provided:

- Call Stack view
- Classes view
- Locals view
- Memory view
- Registers view
- Stack view
- Symbols view
- Watch view.

Unless you are using RealMonitor or an OS-aware plug-in the target must be stopped to see the information in these views.

See also

- *Call Stack view* on page 1-10
- *Classes view* on page 1-12
- *Locals view* on page 1-15
- *Memory view* on page 1-16
- *Registers view* on page 1-19
- *Stack view* on page 1-20
- *Symbols view* on page 1-21
- *Watch view* on page 1-22
- *RealMonitor connections* on page 1-27
- *OS-aware connections* on page 1-27
- *Execution control* on page 1-31
- *Breakpoints in RealView Debugger* on page 1-36
- Chapter 13 *Examining the Target Execution Environment*.

1.10.2 Other methods of examining the target execution environment

You can use CLI commands and macros to examine the values of registers and memory locations. Macros enable you to monitor target execution without stopping the target. For example, you can create a macro that writes the values of specific variables to a user-defined window or file, or to the Code window. You can either run this macro yourself, or set a breakpoint that runs the macro when it is activated.

Unless you are using RealMonitor or an OS-aware plug-in the target must be stopped to examine the values of registers and memory locations.

See also

- *Command scripts* on page 1-42

- *Macros* on page 1-43
- *RealMonitor connections* on page 1-27
- *OS-aware connections* on page 1-27
- *Execution control* on page 1-31
- *Breakpoints in RealView Debugger* on page 1-36
- Chapter 13 *Examining the Target Execution Environment*.

1.11 Altering the target execution environment

When execution stops at specified points, you can alter the values of specific items to influence how the target executes when you next start it.

See also:

- *Debug views*
- *Other methods of altering the target execution environment.*

1.11.1 Debug views

The following debug views enable you to change the values of specific items:

- Call Stack view
- Comms Channel view, for communication over DCC
- Locals view
- Memory view
- Registers view
- Stack view
- Watch view.

Unless you are using RealMonitor or an OS-aware plug-in the target must be stopped to see the information in these views and change any values.

See also

- *Call Stack view* on page 1-10
- *Comms Channel view* on page 1-13
- *Locals view* on page 1-15
- *Memory view* on page 1-16
- *Registers view* on page 1-19
- *Stack view* on page 1-20
- *Watch view* on page 1-22
- *RealMonitor connections* on page 1-27
- *OS-aware connections* on page 1-27
- *Execution control* on page 1-31
- *Breakpoints in RealView Debugger* on page 1-36
- Chapter 14 *Altering the Target Execution Environment*.

1.11.2 Other methods of altering the target execution environment

You can use CLI commands and macros to change the values of registers and memory locations.

Unless you are using RealMonitor or an OS-aware plug-in the target must be stopped to change the values of registers and memory locations.

See also

- *Command scripts* on page 1-42
- *Macros* on page 1-43
- *RealMonitor connections* on page 1-27
- *OS-aware connections* on page 1-27
- *Execution control* on page 1-31
- *Breakpoints in RealView Debugger* on page 1-36

- Chapter 14 *Altering the Target Execution Environment.*

1.12 Command scripts

You can perform many debugging operations using RealView Debugger CLI commands. You can use these commands directly at the RealView Debugger command line, or in command scripts.

You can create command scripts in the following ways:

- using any text editor
- using the logging facility to create a log file containing the CLI commands that are generated by RealView Debugger and the commands that you enter.

You can run scripts in the following ways:

- by specifying the script as an argument to the RealView Debugger command on startup
- by adding the script to the Scripts toolbar, shown Figure 1-23, from which you can run the currently selected command script:



Figure 1-23 Scripts toolbar

- by using the INCLUDE command within RealView Debugger.

When a script is running a Script indicator appears in the Code Window status bar.

See also:

- *Code window status bar* on page 1-5
- *Macros* on page 1-43
- Chapter 15 *Debugging with Command Scripts*.
- Chapter 16 *Using Macros for Debugging*.

1.13 Macros

Macros enable you to perform complex debugging tasks that you cannot do using RealView Debugger CLI commands alone. RealView Debugger provides many predefined macros that you can use directly at the command line, or within your own scripts. In addition, you can define your own user-defined macros. User-defined macros can contain RealView Debugger CLI commands and calls to other macros.

You can either run macros directly, or as part of another operation. For example, you can set a breakpoint that runs macros when it is activated. The macro might test values of variables when the breakpoint is hit, and then cause execution to stop or continue depending on the values of variables.

See also:

- *Breakpoints in RealView Debugger* on page 1-36
- *Command scripts* on page 1-42
- *Chapter 15 Debugging with Command Scripts*
- *Chapter 16 Using Macros for Debugging*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 3 *RealView Debugger Predefined Macros*.

1.14 Log and journal files

The logging and journaling features of RealView Debugger enable you to keep a log of CLI commands, messages output by your application, and other messages displayed in the Output view.

See also:

- *Types of log and journal files.*

1.14.1 Types of log and journal files

RealView Debugger enables you to create the following files:

Log files During your debugging session, you can create a log file containing:

- all the commands you enter
- commands that are generated by the RealView Debugger GUI.

You can then use this file as the basis of a command file or a macro. By default, log files have the extension .log or .inc, but you can use any extension.

STDIO log files

A STDIO log file enables you to keep a record of messages sent to STDIO from the target, that is your application. This might be useful for controlling debugging by running scripts without using the RealView Debugger user interface. By default, these files have the extension .log, but you can use any extension.

Journal files A session journal file that contains all information including:

- commands you enter
- commands that are generated by the RealView Debugger GUI
- any messages displayed in the Output view
- messages from your application.

By default, journal files have the extension .jou, but you can use any extension.

You can open one or all of these types of file in a debugging session. However, only one file of each type can be open.

See also

- *Creating a log file for use as a command script* on page 15-5.

1.15 Editing source files

RealView Debugger automatically opens source files when you:

- Load an image that contains debug symbols. In this case, the source file containing the current scope (the image entry point in this case) is opened.
- Change scope to a location that corresponds with a line in your source code, for example, you set the PC to the start address of a function.
- Execution stops at a location that corresponds with a line in your source code, for example at a breakpoint.

In these cases, the source files are opened in read-only mode. If you attempt to edit a read-only source file, then RealView Debugger prompts you to allow editing.

You can also open source files yourself. In this case, the source files are opened in edit mode.

For full details on editing source files, see the RealView Debugger online help.

See also:

- *Opening source files for a loaded image* on page 4-26.

1.16 Searching source files

RealView Debugger provides various features for searching your source files:

- The Search toolbar in the Code window, shown in Figure 1-24, enables you to perform a simple text search in the source file that you are currently viewing in the Code window.



Figure 1-24 Search toolbar

- A Find in Files dialog box enables you to perform more complex searches on all source files in a specified directory, and subdirectories.

For full details on searching source files, see the RealView Debugger online help.

Chapter 2

The RealView Debugger Environment

This chapter describes how to start RealView® Debugger. It also describes how to set up environment variables and directories to your requirements. It contains the following sections:

- *Starting RealView Debugger from the command line* on page 2-2
- *Starting RealView Debugger after installing other components* on page 2-7
- *Setting user-defined environment variables* on page 2-8
- *Redefining the RealView Debugger directories* on page 2-9.

2.1 Starting RealView Debugger from the command line

If your *RealView Development Suite* (RVDS) environment is correctly configured, you can start RealView Debugger from the command line by typing `rvdebug` together with any arguments.

By default, the RealView Debugger starts in graphical user interface (GUI) mode. The GUI has a built-in command line. To start RealView Debugger in command line interface (CLI) mode, you must specify the `--cmd` option.

————— Note —————

If you are having problems starting RealView Debugger, see the *RealView Development Suite Getting Started Guide* for details of how to fix problems with your RVDS environment.

See also:

- *Syntax of the rvdebug command*
- *Starting multiple instances of RealView Debugger* on page 2-5
- *Examples* on page 2-5.

2.1.1 Syntax of the `rvdebug` command

The syntax for starting RealView Debugger at the command-line is as follows:

```
rvdebug [{--batch | --cmd}] [--cleanstart] [--install=pathname] [--home[=]pathname]
[{-init[=]connection | --target[=]connection}] [{--image[=]pathname |
--exec[=]pathname}]
[{-inc[=]pathname | --script[=]pathname}]
[{-project[=]filename | --no_project}]
[{-workdir[=]pathname | --reinitialize_workdir}]
[{-journal[=]pathname] [--log[=]pathname] [--stdiolog[=]pathname]}
[{-workspace=pathname | --aws=pathname}] [--no_logo]
```

————— Note —————

You can use `--` or `-` when specifying the command line arguments.

The command line arguments are:

- | | |
|----------------------|---|
| <code>--aws</code> | See the description of <code>--workspace</code> . |
| <code>--batch</code> | Start a RealView Debugger session in batch mode, that is without any user interaction.
Use this with <code>--script</code> to run a script file containing commands. |

————— Note —————

If you use `--batch` without `--script`, RealView Debugger displays an error message and exits.

`--cleanstart` Clears the contents of your home directory, or the home directory specified with `--home`, to restore the default set of configuration files.

————— Note —————

All existing files in the directory are deleted before the new configuration files are created.

--cmd	Start a RealView Debugger session in command line mode, where you can use RealView Debugger CLI commands to carry out debugging tasks. This enables you to interact with the debugger without using the RealView Debugger GUI.
--exec	See the description of --image.
--help	Displays a summary of the rvdebug command-line arguments.
--home	<p>Specifies the complete path to your RealView Debugger home directory used for the debugging session. You can optionally include <code>=</code>, for example <code>--home=homedir</code>. If this argument is not specified, your default login name is used.</p> <p>The path you specify does not have to exist. For example, you might want to specify a different path if you want to perform any of the tutorials described in the RealView Debugger documentation, or you want to experiment with different Debug Configurations. In this way, your default home directory is preserved.</p>
	Note
	--home is supported only on Windows.
--inc	See the description of --script.
--image	<p>Specifies the image to be loaded when RealView Debugger runs. You can specify only a single image to load when you start RealView Debugger from the command line. However, after RealView Debugger has started, you can load additional images as required.</p> <p>The image specification can also include section details and image arguments, and has the format:</p> <p style="padding-left: 2em;"><code>--image[= "image[;sections][;arguments]"</code></p> <p>where:</p> <ul style="list-style-type: none"> • <i>image</i> is the image name, and path if required • <i>sections</i> is a comma-separated list of sections • <i>arguments</i> is a space-separated list of arguments • <code>=</code> is optional.
	Note
	You can optionally use --target to specify a connection when loading an image in this way. If you omit --target, then RealView Debugger does one of the following: <ul style="list-style-type: none"> • Uses an existing connection that is saved in the workspace. If more than one connection is saved, then the first connection that is established is used. • Displays a prompt asking if you want to defer the load until a connection is available. If you choose to defer the load, then RealView Debugger attempts to load the image to the first target you connect to.
--init	See the description of --target.
--install	<p>Specifies an installation directory if this differs from the default installation. This must point to the following directory:</p> <p style="padding-left: 2em;"><i>install_directory\RVD\Core\...\\platform</i></p> <p>On Windows systems, this is then used to define the location of the default RealView Debugger home directory when the debugger runs for the first time.</p>

This overrides the environment variable RVDEBUG_INSTALL, and must be used if RVDEBUG_INSTALL is not set.

--journal	Start a RealView Debugger session with the specified journal file open for writing. You can optionally include =, for example --journal= <i>filename</i> . Can be replaced with -jou or -j.
--log	Start a RealView Debugger session with the specified log file open for writing. You can optionally include =, for example --log= <i>filename</i> . Can be replaced with -l.
--no_logo	Start a RealView Debugger session without displaying the splash screen. You can also use --nologo.
--project	Use this in combination with --workdir. Instructs RealView Debugger to load the specified project template file. Options from the project template are set only if they do not conflict with other options specified on the command line.

Note

This option overrides the RVDS_PROJECT environment variable if it is set.

Use --no_project to prevent the use of a project template specified by the RVDS_PROJECT environment variable if it is set.

--reinitialize_workdir

Use this in combination with --workdir. If --workdir refers to an existing working directory that contains modified project template files, then specify this option to delete the working directory and recreate it with new copies of the original project template files.

--script	Start a RealView Debugger session with the specified script file that contains RealView Debugger commands. You can optionally include =.
----------	--

Use --script in:

- Batch mode in association with the --batch setting, to execute the commands contained in the script file and then exit the debugger.
- Command-line mode in association with the --cmd setting, to execute the commands contained in the script file. If the script file does not force RealView Debugger to exit, then the RealView Debugger displays the CLI prompt ready for you to continue debugging.
- GUI mode (that is, without --batch and --cmd), to execute the commands contained in the script file during a debugging session. If the script file does not force RealView Debugger to exit, then the RealView Debugger Code window remains open ready for you to continue debugging.

--stdiolog	Start a RealView Debugger session with the specified STDIO log file open for writing. You can optionally include =, for example --stdiolog= <i>filename</i> . Can be replaced with -s.
------------	--

--target	Specifies the connection to establish when RealView Debugger runs. You can optionally include =, for example --target= <i>connection</i> . This option requires that you specify the connection using the named connection format:
----------	--

@target@DebugConfiguration

For example, to connect to the Cortex-A8 model of the ISSM Debug Configuration in the *Instruction Set System Model* (ISSM) Debug Interface, specify @ARM_Cortex-A8@ISSM.

Note

You can specify only a single target connection. However, after RealView Debugger has started, you can make additional connections as required.

- version Displays a dialog box showing RealView Debugger version information.
- workdir Use this in combination with --project to specify your working directory to use for that project template. Project templates only require working directories if they include private files, such as RealView Debugger configuration files.
If --workdir is required, then an error message is produced if you try to use --project without --workdir.

Note

This option overrides the RVDS_PROJECT_WORKDIR environment variable if it is set.

- workspace Specify --workspace[=]pathname to start a RealView Debugger session with the specified workspace. This overrides any workspace specification that was stored when the previous session ended. If the specified workspace file does not exist, you are prompted to create it:
 - Click **Yes** to create the workspace file and start RealView Debugger.
 - Click **No** to start RealView Debugger with a default workspace.

See also

- *Types of log and journal files* on page 1-44
- *Setting user-defined environment variables* on page 2-8
- *Defining the home directory on Windows* on page 2-10
- *CONNECTION* on page A-14
- Chapter 15 *Debugging with Command Scripts*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 2 *RealView Debugger Commands*.

2.1.2 Starting multiple instances of RealView Debugger

You can start multiple instances of RealView Debugger if required. However, if you try to start a second instance of RealView Debugger that uses the same home directory as the first instance, RealView Debugger displays a warning dialog box. To prevent this, start additional instances of RealView Debugger using --home argument, and specify a different home directory.

See also

- *Syntax of the rvdebug command* on page 2-2
- Chapter 15 *Debugging with Command Scripts*.

2.1.3 Examples

The following examples show how to use some of the command line options when starting RealView Debugger from the command line on Windows. The examples assume that RealView Debugger is installed on the C drive, and use a working directory on the D drive.

- To start RealView Debugger on Windows and specify a home directory, where RVDEBUG_HOME is not set or you want to override it, enter:
`rvdebug.exe --home="D:\rvd_work\home\my_user_name"`

- To start RealView Debugger and specify a workspace, enter:
`rvdebug.exe" --workspace="D:\rvd_work\home\my_user_name\friday_test.aws"`
- To start RealView Debugger with a log file open for writing, enter:
`rvdebug.exe --log="D:\rvd_work\home\my_user_name\test_files\my_log.log"`
- To start RealView Debugger with a connection to an ARM926EJ-S™ processor, and load an image that takes two arguments, enter:
`rvdebug.exe --target=@ARM926EJ-S_0@RVI --image "D:\rvd_work\images\my_image.axf;;100 200"`
- To start RealView Debugger with a working directory C:\myprojects\cp926 and to use the project template for an ARM926EJ-S processor on an ARM Integrator™/CP development board, enter:
`rvdebug.exe --workdir=C:\myprojects\cp926 --project="C:\Program Files\ARM\projects\ARM RealView Development Boards\cp926.xml"`

See also

- *Syntax of the rvdebug command* on page 2-2
- *Starting multiple instances of RealView Debugger* on page 2-5.

2.2 Starting RealView Debugger after installing other components

If you install additional software components after you have used RealView Debugger, for example, the DSTREAM and RealView ICE host software, then the next time you start RealView Debugger it prompts you to update the settings files.

To start RealView Debugger after installing additional software components:

1. Start RealView Debugger in the usual way.

RealView Debugger informs you that it is about to update the configuration files in your default settings directory, shown in Figure 2-1.

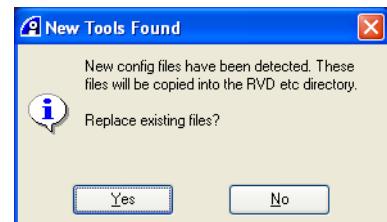


Figure 2-1 Prompt to update configuration files

2. Click **Yes** to update the configuration files, and replace any existing files that exist for previous versions of that component.

————— **Note** —————

If you click **No**, then this prompt is displayed again when you next start RealView Debugger.

RealView Debugger then prompts you to rebuild the rvdebug.brd file, shown in Figure 2-2.

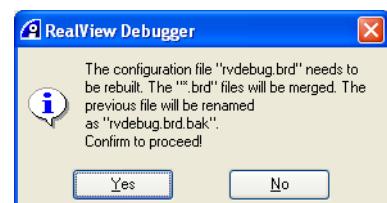


Figure 2-2 Prompt to rebuild the rvdebug.brd file

3. Click **Yes** to rebuild the rvdebug.brd file. You must do this if the installed component software is for a new Debug Interface, such as RealView ICE.

————— **Note** —————

If you click **No**, then RealView Debugger is unable to recognize the installed component. Also, this prompt is not displayed again when you next start RealView Debugger.

RealView Debugger completes the startup procedure.

See also:

- *Files in the default settings directory* on page B-3.

2.3 Setting user-defined environment variables

You can set user-defined environment variables to reconfigure the RealView Debugger environment to your particular requirements.

You might want to do this, for example, if you are working as part of a development team and you:

- share a common RealView Debugger home directory
- have a common set of RealView Debugger target configuration files.

See also:

- *Overriding the default home directory*
- *Specifying a shared location*
- *Specifying the location of a project template.*

2.3.1 Overriding the default home directory

Set RVDEBUG_HOME to override the default home directory, for example:

```
set RVDEBUG_HOME=D:\rvd_work\home
```

— Note —

On Windows, this also overrides the --home command line option to the rvdebug command.

See also

- *Starting RealView Debugger from the command line* on page 2-2
- *Defining the home directory on Windows* on page 2-10.

2.3.2 Specifying a shared location

Set RVDEBUG_SHARE to specify a shared location for RealView Debugger target configuration files:

```
set RVDEBUG_SHARE=H:\ournet\dev\rvd\shared
```

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Preparing Debug Configurations for distribution* on page 3-56
 - *Distributing Debug Configurations to other machines and users* on page 3-58.

2.3.3 Specifying the location of a project template

To use a specific project template, use the following environment variables:

- RVDS_PROJECT to specify the location of a project template file. Use the --project command line argument to override this environment variable.
- RVDS_PROJECT_WORKDIR to specify a working directory to use for the project template specified in RVDS_PROJECT. Use the --workdir command line argument to override this environment variable.

See also

- *Starting RealView Debugger from the command line* on page 2-2.

2.4 Redefining the RealView Debugger directories

RealView Debugger must be able to identify the installation directory and a home directory so that it can locate files and store updated files or user configuration details. This section describes how to redefine the installation and home directories, if required.

See also:

- *Defining the installation directory*
- *Defining the home directory on Windows* on page 2-10
- *The home directory on Red Hat Linux* on page 2-10
- *The current working directory* on page 2-10
- *Using the examples directories* on page 2-11.

2.4.1 Defining the installation directory

RealView Debugger must be able to identify the installation directory so that it can locate user files and configuration files. It uses the following sequence of tests to determine the installation directory:

1. The `--install` command line argument, where used.
2. The `RVDEBUG_INSTALL` environment variable, which is set during installation to:
install_directory\RVD\Core\...\platform

— Note —

If the `RVDEBUG_INSTALL` environment variable is not set, then you must start RealView Debugger with the `--install` option.

If the debugger cannot find the install directory, it terminates.

Shortcuts that are installed on the Windows **Start** menu might include the `--install` option to define the directory. If you did not install the debugger in the default location and you create your own shortcuts, or run `rvdebug.exe` from the command line, you must either include the same `--install` option or define the `RVDEBUG_INSTALL` environment variable globally, for example using the Windows Control Panel.

See also

- *Starting RealView Debugger from the command line* on page 2-2.

2.4.2 Defining the home directory on Windows

RealView Debugger requires a home directory to store user-specific settings and configuration files. This is not the same as your Windows home directory.

On Windows, the location of this directory depends on the environment variables set, and the command line arguments used, when RealView Debugger starts. It uses the following sequence of tests to determine the home directory:

1. The --home argument when starting RealView Debugger from the command line to specify an explicit path, for example:

```
--home="C:\rvd_work\home\user_name"
```

The user name must not contain spaces. Any user name longer than 11 characters is automatically truncated.

2. The RVDEBUG_HOME environment variable to use a directory other than the default home directory, for example:

```
set RVDEBUG_HOME=D:\rvd_work\home\user_name
```

The user name must not contain spaces. Any user name longer than 11 characters is automatically truncated.

3. Your default Windows login user ID, for example:

```
C:\Documents and Settings\userID\Application Data\ARM\rvdebug\version.
```

This is the default home directory that is used by RealView Debugger.

See also

- *Starting RealView Debugger from the command line* on page 2-2
- *Overriding the default home directory* on page 2-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Locating the configuration files* on page 1-16.

2.4.3 The home directory on Red Hat Linux

On Red Hat Linux, your home directory is in ~/rvd. You cannot change this directory.

2.4.4 The current working directory

RealView Debugger keeps track of the current working directory. The directory depends on where you first start RealView Debugger.

When you load an image, RealView Debugger changes the directory to the location of the image. However, the location does not change if you load a binary.

Identifying the location of the current working directory

To identify the location of the current working directory, enter the PWD CLI command.

Changing the current working directory

To change the current working directory, use the CWD command.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the CWD and PWD commands.

2.4.5 Using the examples directories

This section describes the various demonstration projects that are supplied as part of the RVDS root installation.

RealView Development Suite example sources and images

The RVDS root installation contains programs in the form of ARM® assembly language, C, or C++ source code files. Some prebuilt images are also provided. These projects are stored in:

install_directory\RVDS\Examples\...\main\project

Flash programming examples

The RealView Debugger root installation includes demonstration projects and associated files for programming selected Flash devices. These are in:

install_directory\RVD\Flash\...\platform.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger*.
- *RealView Development Suite Getting Started Guide*.

Chapter 3

Target Connection

This chapter describes how to connect to and disconnect from a target in RealView® Debugger. It contains the following sections:

- *About target connection* on page 3-2
- *About creating a Debug Configuration* on page 3-8
- *Changing the name of a Debug Configuration* on page 3-17
- *Copying an existing Debug Configuration* on page 3-18
- *Deleting a Debug Configuration* on page 3-19
- *Customizing a Debug Configuration* on page 3-20
- *Connecting to a target* on page 3-27
- *Viewing information about the target topology* on page 3-36
- *Setting top of memory for the current debugging session* on page 3-38
- *Viewing connection details* on page 3-40
- *Connecting to a target on startup* on page 3-42
- *Connecting to a target using different modes* on page 3-43
- *Connecting to multiple targets* on page 3-46
- *Connecting to all targets for a specific Debug Configuration* on page 3-48
- *Changing the current target connection* on page 3-50
- *Disconnecting from a target* on page 3-52
- *Disconnecting from a target using different modes* on page 3-54
- *Disconnecting from multiple targets* on page 3-56
- *Disconnecting from all targets for a specific Debug Configuration* on page 3-58
- *Storing connections when exiting RealView Debugger* on page 3-59
- *Troubleshooting target connection problems* on page 3-60.

3.1 About target connection

RealView Debugger works in conjunction with either a hardware or a software debug target. The following sections introduce the features of RealView Debugger that enable you to connect to your debug targets:

- *The Connect to Target window*
- *Target Connections* on page 3-4
- *Connections to multiple targets* on page 3-5
- *Connections to trace components* on page 3-5
- *Connections in the Home Page* on page 3-7.

3.1.1 The Connect to Target window

The Connect to Target window in RealView Debugger lists the Debug Interfaces that are currently installed on your workstation. An example Connect to Target window is shown in Figure 3-1:

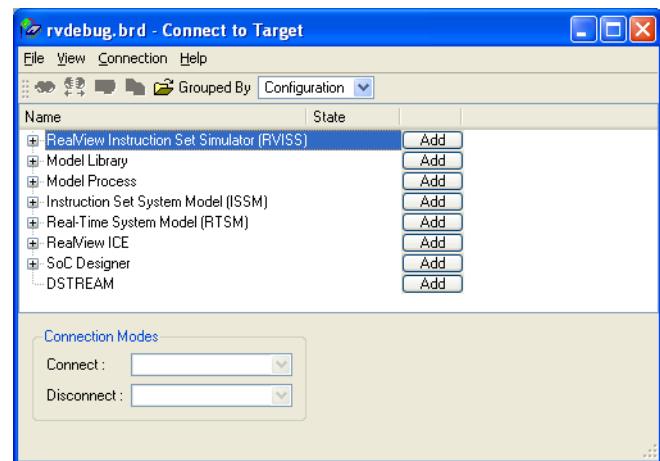


Figure 3-1 Example Connect to Target window

You can use the Connect to Target window to perform all tasks related to target connection.

Note

RealView Debugger remembers the last 10 target connections that you made. These connections are listed in the **Home Page** tab of the Code window, and in the **Recent Connections** menu that is available from the **Target** menu.

Debug Interfaces

The following Debug Interfaces are available:

Hardware Debug Interfaces

The following software Debug Interfaces are available:

- The DSTREAM Debug Interface is available if installed, which enables you to debug hardware through a DSTREAM unit.
- The RealView ICE Debug Interface is available if installed, which enables you to debug hardware through a RealView ICE unit.

Note

This document assumes that you have installed the DSTREAM and RealView ICE host software, and updated the firmware as required. However, you must purchase the DSTREAM unit or RealView ICE unit separately.

Software Debug Interfaces

The following software Debug Interfaces are available:

- Instruction Set System Model (ISSM), to connect to simulated Cortex™ processors. ISSM is always installed with RealView Debugger.
- Model Library, to connect to a *Cycle Accurate Debug Interface* (CADI) model defined in a model library file.
- Model Process, to connect to a CADI model that is currently running, such as an Exported Virtual System.
- Real-Time System Model (RTSM), to connect to models of real-time systems.

Note

Be aware that *Emulation Baseboard* (EB) RTSMs are not intended to be software implementations of particular revisions of EB hardware.

- RealView ARMulator ISS, to connect to *RealView ARMulator® ISS* (RVISS) simulated processors. RVISS is always installed with RealView Debugger.
- SoC Designer, to connect to Carbon SoC Designer Plus models. You must purchase the Carbon SoC Designer Plus application separately.

Target groupings

Each Debug Interface contains one or more Debug Configurations. A Debug Configuration identifies the targets that are available on the associated development platform. You can display the targets using the following groupings:

Target All the targets are shown as a single list in each Debug Interface. The targets are listed in the order of the associated Configuration name, shown in Figure 3-2:

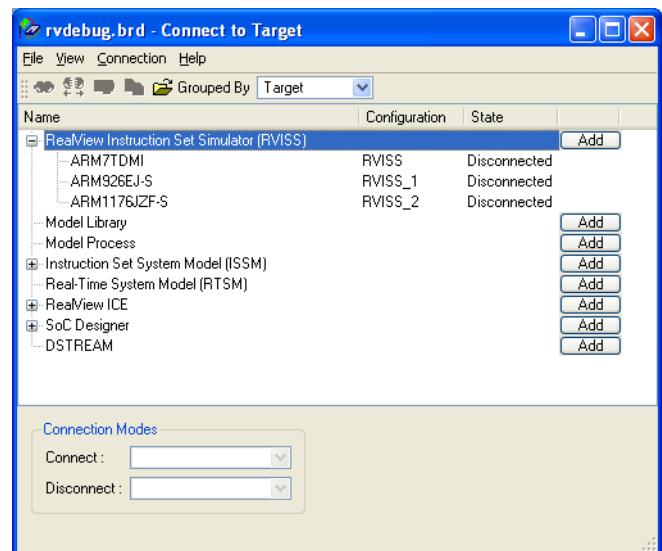


Figure 3-2 Example Connect to Target window (Target group)

Configuration

The targets are listed under the name of the associated Debug Configuration, shown in Figure 3-3:

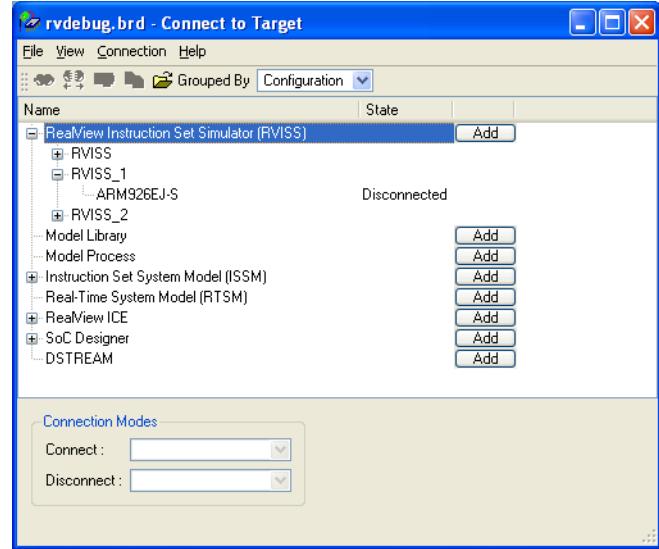


Figure 3-3 Example Connect to Target window (Configuration group)

You can have multiple Debug Configurations for the same development platform. If you have multiple debugging platforms, then you must create a separate Debug Configuration for each platform. A Debug Configuration enables you to set up a custom debugging environment

Additional connection features

The Connect to Target window also contains additional features that enable you to:

- configure a selected Debug Interface
- connect and disconnect all targets for a selected Debug Configuration
- connect and disconnect using different modes.

See also

- *About creating a Debug Configuration* on page 3-8
- *Making a connection from the Home Page* on page 3-28
- *Connecting to a target using different modes* on page 3-43
- *Connecting to all targets for a specific Debug Configuration* on page 3-48
- *Disconnecting from all targets for a specific Debug Configuration* on page 3-58
- *Disconnecting from a target using different modes* on page 3-54
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 2 *Customizing a Debug Interface configuration*.

3.1.2 Target Connections

Connecting to a target requires that you first add an appropriate Debug Configuration to identify the targets on your development platform. However, default Debug Configurations are provided with RVDS. You can connect to these without having to configure them.

You can create separate Debug Configurations for the same target, with each configuration specifying a different debugging environment.

See also

- *Default Debug Configurations provided with RVDS on page 1-24*
- *About creating a Debug Configuration on page 3-8*
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 2 *Customizing a Debug Interface configuration*.

3.1.3 Connections to multiple targets

You can connect to multiple targets, which can be available through the same Debug Interface, such as RealView ICE, or through multiple Debug Interfaces, for example, RealView ICE and Real-Time System Model (RTSM).

Although you can connect to and disconnect from multiple targets individually, if all the targets are available in the same Debug Configuration, you can connect to and disconnect from all the targets in a single operation.

Also, there are special features available in RealView Debugger that enable you to debug multiprocessor development platforms.

See also

- *Connecting to multiple targets on page 3-46*
- *Connecting to all targets for a specific Debug Configuration on page 3-48*
- *Changing the current target connection on page 3-50*
- *Disconnecting from all targets for a specific Debug Configuration on page 3-58*
- Chapter 7 *Debugging Multiprocessor Applications*.

3.1.4 Connections to trace components

If your development platform has a simple configuration containing trace components, then RealView Debugger automatically connects to the trace components when you connect to the target processor. A simple configuration is assumed to be a single ARM architecture-based processor with an *Embedded Trace Macrocell™* (ETM™), and optionally an *Embedded Trace Buffer™* (ETB™).

Note

Except for setting up the conditions for trace capture, no additional trace hardware configuration is required for a simple configuration.

By default, all trace components are hidden in the Connect to Target window, and only the target processors are visible.

For a development platform that has a more complex configuration, then how you configure it depends on your requirements:

- For a CoreSight™ development platform:
 1. Autoconfigure the Debug Interface configuration to detect the CoreSight *Debug Access Port* (DAP).
 2. If your CoreSight DAP contains a ROM table, then read the ROM table to determine the target processor, and the remaining CoreSight components. Otherwise, you must manually add the remaining devices.
 3. Create a CoreSight associations file, and assign it to the Debug Interface configuration.
 4. Connect to all CoreSight components.

Note

All CoreSight components must remain connected to capture trace.

5. Set up the trace hardware configuration for each CoreSight component.

Note

If you do not want to capture trace information, then you have only to add the CoreSight DAP and the target processor.

- For a non-CoreSight development platform:
 1. Autoconfigure to detect the target processors, and trace components.
 2. Create an associations file, and assign it to the Debug Interface configuration.
 3. Set up the trace hardware configuration for each trace component.

Supported CoreSight components

Table 3-1 lists the CoreSight components supported in this release, and the corresponding connection name that appears in the Connect to Target window. However, you can specify different names for the components in the CoreSight associations file.

Table 3-1 Connection names for supported CoreSight components

CoreSight component	Default connection name
JTAG Access Port for connection to an ARM1136JF-S processor	ARM1136JFS-JTAG-AP
JTAG Access Port for connection to an ARM1156T2F-S processor	ARM1156T2FS-JTAG-AP
JTAG Access Port for connection to an ARM1176JZ-F processor	ARM1176JZF-JTAG-AP
<i>AHB Trace Macrocell (HTM)</i> ^a	CSHTM
<i>Cross Trigger Interface (CTI)</i>	CSCTI
<i>Embedded Trace Buffer (ETB)</i>	CSETB
<i>Embedded Trace Macrocell (ETM)</i>	CSETM
<i>Instrumentation Trace Macrocell (ITM)</i> ^a	CSITM
<i>Program Flow Trace Macrocell (PTM)</i> ^a	CSPTM
<i>Serial Wire Output (SWO)</i> ^a	CSSWO
Trace Funnel	CSTFunnel
<i>Trace Port Interface Unit (TPIU)</i>	CSTPIU

a. Although this component can be added as a target, you cannot capture trace from it in this release.

Considerations for connections to CoreSight components

Be aware of the following for connections to trace components:

- If any of the following components are required by your development platform, then connections for these components do not appear in the Connect to Target window:
 - CoreSight DAP (ARMCS-DP)
 - Serial Wire Debug DAP (ARMSW-DP)

- JTAG-DAP (ARMJTAG-DP).

You must use the DSTREAM and RealView ICE RVConfig utility to configure these components.

- Only the Registers and Memory views are supported in the Code window when a CoreSight component connection is currently displayed in that window.
- You can configure a CoreSight component using:
 - the Hardware Configuration dialog for that component, which is the recommended method
 - the Registers view.
- You can set up associations for the CoreSight components to give a topology view. You assign the associations using the RVConfig utility.

See also

- *Viewing information about the target topology* on page 3-36
- *Configuring embedded cross-triggering* on page 7-25
- *Configuring CoreSight embedded cross-triggering* on page 7-27
- *Changing the value of a register* on page 14-3
- the following in the *RealView Debugger Trace User Guide*:
 - Chapter 4 *Configuring the ETM*
 - *Configuring trace capture from a CoreSight ETM with RealView Trace* on page B-5
 - *Configuring trace capture from a CoreSight ETB with DSTREAM and RealView ICE* on page B-7.
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

3.1.5 Connections in the Home Page

When you establish a connection in a RealView Debugger session, an entry for the connection is placed on the **Home Page** in the Code window. The **Home Page** stores the last 10 connections you have established. You can establish a connection by clicking on a connection entry in the **Home Page**.

See also

- *Making a connection from the Home Page* on page 3-28.

3.2 About creating a Debug Configuration

Before you can connect to the targets on your development platform, you must create a Debug Configuration for the required Debug Interface.

See also:

- [Adding a new Debug Configuration](#)
- [Creating a DSTREAM or RealView ICE Debug Configuration](#) on page 3-10
- [Creating an RVISS Debug Configuration](#) on page 3-12
- [Creating an ISSM Debug Configuration](#) on page 3-13
- [Creating an RTSM Debug Configuration \(RTSM not running\)](#) on page 3-14
- [Creating an RTSM Debug Configuration \(RTSM running\)](#) on page 3-15
- [Creating a SoC Designer Debug Configuration \(SoC Designer not running\)](#) on page 3-15
- [Creating a SoC Designer Debug Configuration \(SoC Designer running\)](#) on page 3-16.

3.2.1 Adding a new Debug Configuration

To add a new Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.

————— **Note** —————

It is recommended that you add and configure a Debug Configuration using the **Configuration** grouping.

3. Click the **Add** button for the required Debug Interface:
 - **RealView Instruction Set Simulator (RVISS)**
 - **Instruction Set System Model (ISSM)**
 - **Real-Time System Model (RTSM)**
 - **RealView ICE**
 - **SoC Designer**.

The configuration dialog box for the selected Debug Interface is displayed.

4. Configure the Debug Interface targets as required.
5. After you have configured and saved the Debug Interface configuration, a new Debug Configuration is added to that Debug Interface. RealView Debugger assigns a default name to the Debug Configuration according to the Target Interface. Table 3-2 shows the default names.

Table 3-2 Debug Interface configuration names and file extensions

Debug Interface	Default name	Debug Interface configuration file extension
DSTREAM	DSTREAM	.rvc
Instruction Set System Model (ISSM)	ISSM	.smc
Model Library	Library	.cm1
Model Process	Process	.cmp

Table 3-2 Debug Interface configuration names and file extensions (continued)

Debug Interface	Default name	Debug Interface configuration file extension
Real-Time System Model (RTSM)	RTSM	.smc
RealView Instruction Set Simulator (RVISS)	RVISS	.auc
RealView ICE	RVI	.rvc
SoC Designer	SoC	.smc

If a Debug Configuration already exists with the default name, then a numerical suffix *_n* is added to the name, starting at one, for example RVISS_1.

The Debug Configuration name is user-definable, so that you can change it to a more meaningful name.

In addition, a unique Debug Interface configuration file is created for the Debug Configuration. The file has the same name as the Debug Configuration and the suffix *_0.ext*, where *ext* is the file extension appropriate to the Debug Interface (see Table 3-2 on page 3-8). For example, for the Debug Configuration RVISS_1, the Debug Interface configuration file is called RVISS_1_0.auc.

Note

Do not use the same Debug Interface configuration file for more than one Debug Configuration. This is because deleting a Debug Configuration also deletes the configuration file.

6. Optionally, configure the new configuration as required.

After adding the new Debug Configuration, you can customize it if required.

See also

- *Creating a DSTREAM or RealView ICE Debug Configuration* on page 3-10
- *Creating a Model Library Debug Configuration* on page 3-11
- *Creating a Model Process Debug Configuration* on page 3-12
- *Creating an RVISS Debug Configuration* on page 3-12
- *Creating an ISSM Debug Configuration* on page 3-13
- *Creating an RTSM Debug Configuration (RTSM not running)* on page 3-14
- *Creating an RTSM Debug Configuration (RTSM running)* on page 3-15
- *Creating a SoC Designer Debug Configuration (SoC Designer not running)* on page 3-15
- *Creating a SoC Designer Debug Configuration (SoC Designer running)* on page 3-16
- *Changing the name of a Debug Configuration* on page 3-17
- the following in the *RealView Debugger Target Configuration Guide*:
 - *About customizing a DSTREAM or RealView ICE Debug Interface configuration* on page 2-3
 - *Customizing an RVISS Debug Interface configuration* on page 2-11

- *Customizing an ISSM Debug Interface configuration* on page 2-15
- *Customizing an RTSM Debug Interface configuration* on page 2-17
- *Customizing a SoC Designer Debug Interface configuration* on page 2-23
- *Chapter 3 Customizing a Debug Configuration.*
- the following in the *RealView Debugger RTOS Guide*:
 - *Chapter 2 Configuring OS-aware Connections*
- the chapter that describes configuring a debug hardware connection in the *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*
- *RealView ARMulator ISS User Guide*
- *RealView Development Suite Real-Time System Model User Guide*
- Carbon SoC Designer Plus, <http://carbondesignsystems.com/Products/SoCDesigner.aspx>.

3.2.2 Creating a DSTREAM or RealView ICE Debug Configuration

The difference between creating a Debug Configuration for use with DSTREAM or RealView ICE is the Debug Interface that you use. The following procedure uses the RealView ICE Debug Interface as an example.

To create a Debug Configuration for target connection through a RealView ICE unit:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the RealView ICE Debug Interface. The RVConfig utility is displayed that includes a list of the debug interface hardware units available on your local network.
4. Connect to the required debug interface hardware unit, and set up a configuration as required.

If you are configuring a CoreSight system manually, make sure you set up the Trace Associations. The CoreSight devices and associations must be in the correct order to ensure tracing with RealView Debugger.

If you have added the devices to the scan chain yourself, or re-ordered the devices, you might be asked to confirm whether you want to have the devices re-ordered automatically, as shown in the following dialog box:



Figure 3-4 Trace warning dialog box

Select **Yes** to have the RVConfig utility re-order the devices for you.

Select **No** to continue to the RVConfig utility. Your associations have been completed as required, however, because no re-ordering has been made, your associations are not compatible with your debugger. You must fix the ordering and associations before you can use this Debug Configuration in RealView Debugger.

5. Select **Save** from the **File** menu to save the configuration.
6. Select **Exit** from the **File** menu to close the RVConfig utility. The new Debug Configuration is added to the RealView ICE Debug Interface.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing a DSTREAM or RealView ICE Debug Interface configuration for non-CoreSight development platforms* on page 2-7
 - *Customizing a DSTREAM or RealView ICE Debug Interface configuration for development platforms containing CoreSight components* on page 2-9
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

3.2.3 Creating a Model Library Debug Configuration

To create a Model Library Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the Model Library Debug Interface. The Model Configuration Utility dialog box is displayed. Figure 3-5 shows an example:

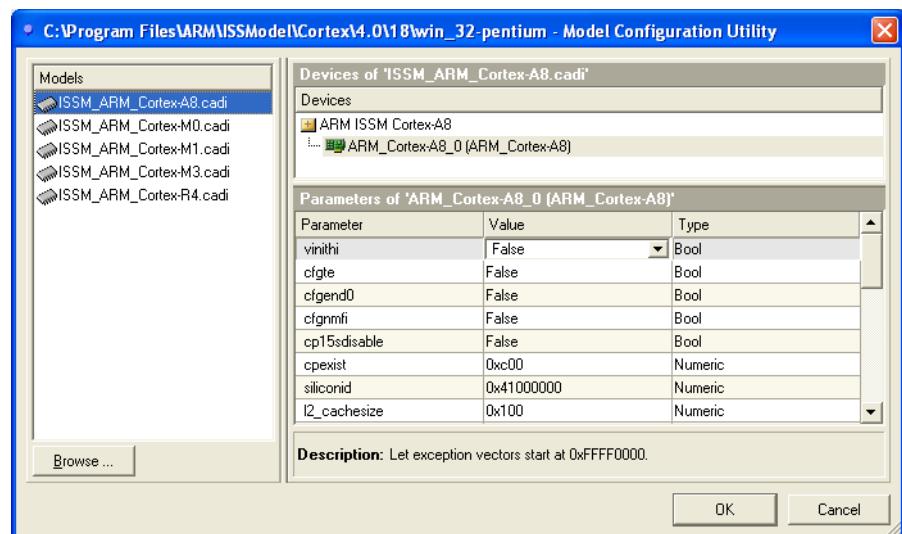


Figure 3-5 Model Configuration Utility dialog box (ISSM)

4. RealView Debugger looks for the ISSM CADI models in the location defined by the ARMROOT environment variable. If you want to locate your own CADI models:
 - a. Click **Browse...** to display the Browse for Folder dialog box.
 - b. Locate the directory containing your models.
 - c. Click **OK**.

The models are listed in the Models pane.

5. Select the required model in the Models pane.
6. Configure the settings as required for each device in the CADI model in the Parameters of '*device_name*' pane.
7. Click **OK** to save your changes and close the dialog box. The Model Configuration Utility dialog box closes, and the new Debug Configuration is added to the Model Library Debug Interface.

The new configuration has the name *library_n*.

See also

- *Creating a Model Process Debug Configuration*
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing a Model Library Debug Interface configuration* on page 2-19.

3.2.4 Creating a Model Process Debug Configuration

To create a Model Process Debug Configuration:

1. Start your model running. This can be as standalone executable or within a model simulator.
2. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
3. Select **Configuration** from the Grouped By drop-down list.
4. Click **Add** for the Model Process Debug Interface. The Model Configuration Utility dialog box is displayed. Any running models are displayed.
5. Select the required model in the Models pane.
6. Configure the settings as required for each device in the CADI model in the Parameters of '*device_name*' pane.
7. Click **OK** to save your changes and close the dialog box. The Model Configuration Utility dialog box closes, and the new Debug Configuration is added to the Model Process Debug Interface.

The new configuration has the name *process_n*.

Use a Model Process Debug Configuration to configure an *Exported Virtual System* (EVS).

See also

- *Creating a Model Library Debug Configuration* on page 3-11
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing a Model Process Debug Interface configuration* on page 2-21.

3.2.5 Creating an RVISS Debug Configuration

To create an RVISS Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.

2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the RealView Instruction Set Simulator (RVISS) Debug Interface. The ARMulator Configuration dialog box is displayed.
4. Select the processor you want to simulate, and configure any other settings as required.
5. Click **OK** to close the ARMulator Configuration dialog box. The new Debug Configuration is added to the RealView Instruction Set Simulator (RVISS) Debug Interface.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing an RVISS Debug Interface configuration* on page 2-11
- *RealView ARMulator ISS User Guide*.

3.2.6 Creating an ISSM Debug Configuration

To create an ISSM Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the Instruction Set System Model (ISSM) Debug Interface. The Model Configuration Utility dialog box is displayed. Figure 3-6 shows an example:

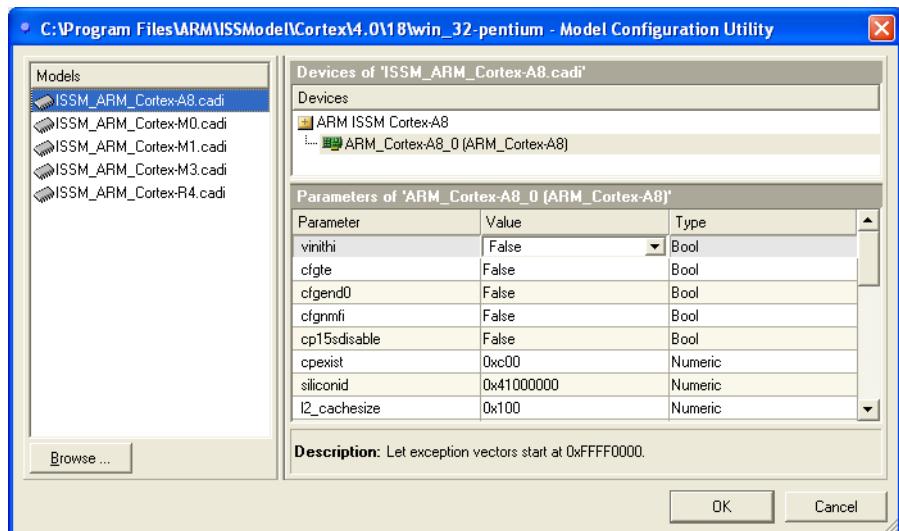


Figure 3-6 Model Configuration Utility dialog box (ISSM)

4. The ISSMs provided with RVDS are listed by default. If you want to locate other ISSMs:
 - a. Click **Browse...** to display the Browse for Folder dialog box.
 - b. Locate the directory containing your models.
 - c. Click **OK**.

The models are listed in the Models pane.
5. Select the required model in the Models pane.

6. Configure the settings as required for each device in the ISSM model in the Parameters of '*device_name*' pane.
 7. Click **OK** to save your changes and close the dialog box. The Model Configuration Utility dialog box closes, and the new Debug Configuration is added to the Instruction Set System Model (ISSM) Debug Interface.
- The new configuration has the name **ISSM_n**.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing an ISSM Debug Interface configuration* on page 2-15.

3.2.7 Creating an RTSM Debug Configuration (RTSM not running)

To create an RTSM Debug Configuration when an RTSM is not currently running:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the Real-Time System Model (RTSM) Debug Interface. The Model Configuration Utility dialog box is displayed. Figure 3-7 shows an example:

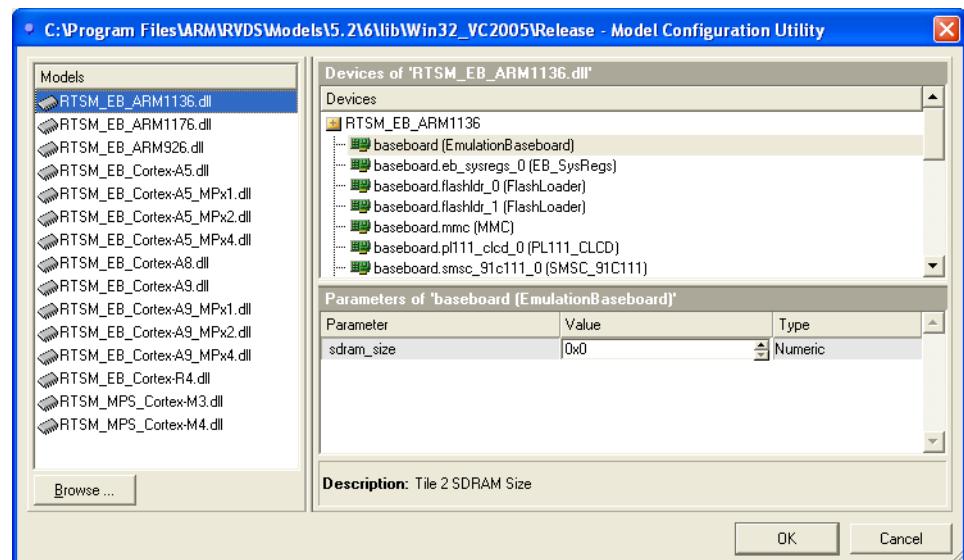


Figure 3-7 Model Configuration Utility dialog box (RTSM)

4. The models provided with RVDS Professional edition are listed in the Models pane. If you want to locate RTSMs you have created with RealView System Generator:
 - a. Click **Browse...** to display the Browse for Folder dialog box.
 - b. Locate the directory containing your models.

Note

The RTSMs provided with RVDS Professional edition are located in:
install_directory\RVDS\Models\...\...\lib\...

- c. Click **OK**. The RTSMs are listed in the Models pane.

5. Select the required RTSM in the Models pane.
6. Configure the settings as required for each device in the model in the Parameters of '*device_name*' pane.
7. Click **OK** to save your changes and close the dialog box. The Model Configuration Utility dialog box closes, and the new Debug Configuration is added to the Real-Time System Model (RTSM) Debug Interface.

The new configuration has the name `RTSM_n`.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing an RTSM Debug Interface configuration* on page 2-17
- *RealView Development Suite Real-Time System Model User Guide*.

3.2.8 Creating an RTSM Debug Configuration (RTSM running)

To create an RTSM Debug Configuration when an RTSM is running:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the Model Process Debug Interface. The Model Configuration Utility dialog box is displayed containing a list of the devices for the running RTSM.
4. Click **OK**. The Model Configuration Utility dialog box closes, and the new Debug Configuration is added to the Model Process Debug Interface.

The Debug Configuration name has the form `process_n`. You might want to rename the configuration to identify it as an RTSM configuration.

————— Note —————

Be aware that when you create a Debug Configuration for a running RTSM you cannot configure the individual devices for that RTSM in RealView Debugger.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing an RTSM Debug Interface configuration* on page 2-17
- *RealView Development Suite Real-Time System Model User Guide*.

3.2.9 Creating a SoC Designer Debug Configuration (SoC Designer not running)

To create a SoC Designer Debug Configuration when a SoC Designer session is not currently running:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the SoC Designer Debug Interface. The Select SoC Designer Session dialog box is displayed.

4. Click **Launch a New SoC Designer Session**. RealView Debugger launches Carbon SoC Designer Simulator.
5. Open your SoC Designer model in Carbon SoC Designer Simulator.
If the Select Application Files dialog box is displayed, load the required images onto each component listed. This is optional, and is not required to create the Debug Configuration.
6. In the Select SoC Designer Session dialog box, click **Scan**. A list of the devices in your model is displayed.
7. Continue at the next step.
8. Click **OK**. The Select SoC Designer Session dialog box closes, and the new Debug Configuration is added to the SoC Designer Debug Interface.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing a SoC Designer Debug Interface configuration* on page 2-23
- Carbon SoC Designer Plus, <http://carbondesignsystems.com/Products/SoCDesigner.aspx>.

3.2.10 Creating a SoC Designer Debug Configuration (SoC Designer running)

To create a SoC Designer Debug Configuration when a SoC Designer session is running:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Click **Add** for the SoC Designer Debug Interface. The Select SoC Designer Session dialog box is displayed containing a list of the components in your SoC Designer model.
4. If multiple SoC Designer sessions are running, select the required session.
5. Click **OK**. The Select SoC Designer Session dialog box closes, and the new Debug Configuration is added to the SoC Designer Debug Interface.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Customizing a SoC Designer Debug Interface configuration* on page 2-23
- Carbon SoC Designer Plus, <http://carbondesignsystems.com/Products/SoCDesigner.aspx>.

3.3 Changing the name of a Debug Configuration

When you create a Debug Configuration, RealView Debugger assigns a default name based on the parent Debug Interface. You can change this to a more meaningful name.

Note

You cannot rename a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

To change the name of a Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Debug Interface containing the Debug Configuration to be renamed.
4. Make sure that there are no targets connected on the Debug Configuration.
5. Right-click on the required Debug Configuration to display the context menu.
6. Select **Rename Configuration** from the context menu.
7. Enter the new name for the Debug Configuration.

For example, you might use the name RVI-AP-ARM966E-S_x3 for a Debug Configuration that comprises three ARM966E-S™ processors and an ARM Integrator™/AP development board, that is accessed through DSTREAM or RealView ICE.

See also:

- *Considerations when renaming a Debug Configuration*
- *Copying an existing Debug Configuration* on page 3-18
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 3 *Customizing a Debug Configuration*.

3.3.1 Considerations when renaming a Debug Configuration

Be aware of the following renaming a Debug Configuration:

- The name must not include spaces. Use either a hyphen (-) or an underscore (_) character instead of a space.
- The name must begin with a letter. RealView Debugger prevents you from entering any other character at the start of the name.
- If you press Delete with the cursor positioned at the start of the current name, then you can delete only those letters up to the one that precedes a non-letter character. For example, if the current name is ARM940T_AP, then you can delete the letters A and R, but not the letter M.

See also

- *Changing the name of a Debug Configuration*.

3.4 Copying an existing Debug Configuration

You might want to create a Debug Configuration that is based on an existing Debug Configuration, for example, to perform OS-awareness debugging.

————— Note —————

You cannot copy a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

To copy an existing Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Debug Interface containing the Debug Configuration to be copied.
4. Make sure that there are no targets connected on the Debug Configuration.
5. Right-click on the required Debug Configuration to display the context menu.
6. Select **Copy Configuration** from the context menu. A new Debug Configuration is created by prefixing the original configuration name with `copy_of_`. For example, if your original configuration was called `RTSM`, then the new configuration name is `copy_of_RTSM`. Also, the Debug Interface configuration file of the original configuration is copied to a new file with the following name:

originalname_n.ext

For example, if the original file is called `rtsm_0.rvc`, the new file might be called `rtsm_0_0.rvc`.

————— Note —————

Do not use the same Debug Interface configuration file for more than one Debug Configuration. This is because deleting a Debug Configuration also deletes the configuration file.

7. Rename the new configuration, if required.
8. Configure the new configuration as required.

See also:

- *Changing the name of a Debug Configuration* on page 3-17
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 3 *Customizing a Debug Configuration*.

3.5 Deleting a Debug Configuration

If you no longer require a Debug Configuration, you can delete it.

— Note —

You cannot delete a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

— Note —

If you want to keep the settings for a Debug Configuration, then you do not have to delete that configuration. Instead, you can hide the configuration so that it is not visible in the Connect to Target window.

To delete a Debug Configuration:

1. Select **Configuration** from the Grouped By drop-down list.
2. Expand the Debug Interface containing the Debug Configuration to be deleted.
3. Make sure that there are no targets connected on the Debug Configuration.
4. Right-click on the required Debug Configuration to display the context menu.
5. Select **Delete Configuration...** from the context menu.
6. Click **OK** when prompted to confirm the deletion. The Debug Configuration is deleted.

— Note —

This also deletes the corresponding CONNECTION= group in the Connection Properties and the related Debug Interface configuration file.

See also:

- the following in the *RealView Debugger Target Configuration Guide*:
 - *What the configuration files contain* on page 1-14
 - *Hiding a Debug Configuration* on page 3-18.

3.6 Customizing a Debug Configuration

Use the Connection Properties dialog box to configure the more commonly used settings in a Debug Configuration.

Before you customize a Debug Configuration, do the following:

1. Select **Connect to Target** from the **Target** menu. The Connect to Target window is displayed.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Debug Interface containing the Debug Configuration to be customized.
4. Make sure that there are no targets connected on the Debug Configuration.

———— Note ————

You cannot customize a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

See also:

- *Displaying the Connection Properties dialog box* on page 3-21
- *Configuring semihosting* on page 3-21
- *Configuring vector catch* on page 3-22
- *Configuring endianness* on page 3-22
- *Configuring top of memory* on page 3-23
- *Enabling RealMonitor* on page 3-23
- *Assigning one or more board/chip definitions* on page 3-23
- *Viewing a board/chip definition* on page 3-24
- *Creating advanced configurations* on page 3-25.

3.6.1 Displaying the Connection Properties dialog box

To display the Connection Properties dialog box:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed. Figure 3-8 shows an example:

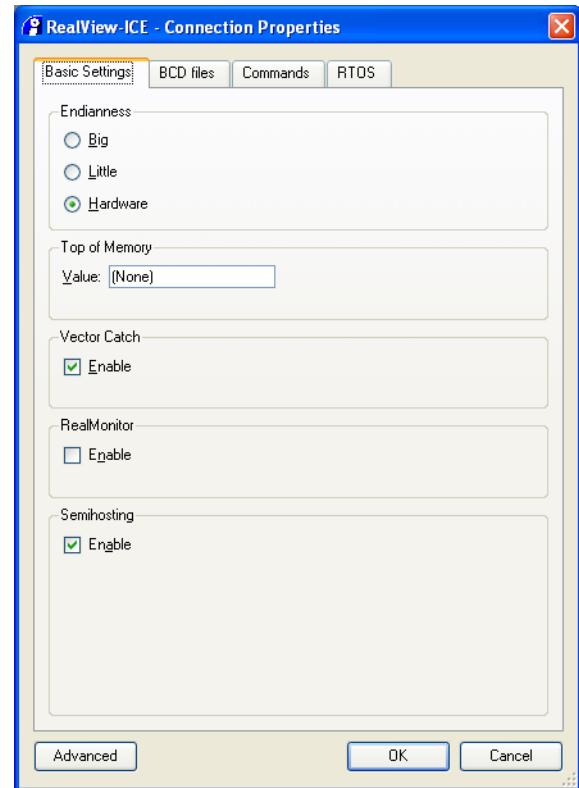


Figure 3-8 Connection Properties dialog box

See also

- *Configuring semihosting*
- *Configuring vector catch* on page 3-22
- *Configuring endianness* on page 3-22
- *Configuring top of memory* on page 3-23
- *Enabling RealMonitor* on page 3-23
- *Assigning one or more board/chip definitions* on page 3-23
- *Viewing a board/chip definition* on page 3-24
- *Creating advanced configurations* on page 3-25.

3.6.2 Configuring semihosting

To configure semihosting for a Debug Configuration:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.

3. On the **Basic Settings** tab, set the **Enable** checkbox to the required state in the Semihosting group.
4. Click **OK** to save your changes.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring semihosting* on page 3-21
 - *ARM_config settings for a Debug Configuration* on page A-11.

3.6.3 Configuring vector catch

To configure vector catch for a Debug Configuration:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
3. On the **Basic Settings** tab, set the **Enable** checkbox to the required state in the Vector Catch group.
4. Click **OK** to save your changes.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring vector catch*
 - *ARM_config settings for a Debug Configuration* on page A-11.

3.6.4 Configuring endianness

To configure endianness for a Debug Configuration:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
3. On the **Basic Settings** tab, select the appropriate radio button in the Endianness group:
 - big
 - little
 - hardware.
4. Click **OK** to save your changes.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21

- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Endianness* on page A-19.

3.6.5 Configuring top of memory

To configure top of memory for a Debug Configuration:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
3. On the **Basic Settings** tab, enter the address of top of memory in the Value field of the Top of Memory group.
For example, enter **0x80000**.
4. Click **OK** to save your changes.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Setting top of memory* on page 4-24
 - *ARM_config settings for a Debug Configuration* on page A-11.

3.6.6 Enabling RealMonitor

To enable RealMonitor for a Debug Configuration:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
3. On the **Basic Settings** tab, click the **Enable** checkbox in the RealMonitor group.
4. Click **OK** to save your changes.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43
 - *Monitor* on page A-15.

3.6.7 Assigning one or more board/chip definitions

To assign one or more board/chip definitions to a Debug Configuration:

1. Right-click on the required Debug Configuration to display the context menu.

2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
3. Click the **BCD files** tab to display the available definitions and assignments.
4. To assign a board/chip definition to this configuration:
 - a. Select the required definition in the Available Definitions list.
 - b. Click the **Add** button.
The selected definition is moved from the Available Definitions list to the Assigned Definitions list.
 - c. Repeat these steps to assign more definitions.
5. Click **OK** to save your changes.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*
 - *BoardChip_name* on page A-9.

3.6.8 Viewing a board/chip definition

To view a board/chip definition:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
3. Click the **BCD files** tab to display the available definitions and assignments.
4. To view a board/chip definition:
 - a. Select the required definition in the appropriate list.
 - b. Click the **View Definition...** button.
The Connection Properties window is displayed with the board/chip group selected for the chosen definition.
 - c. Right-click on the selected group in the left pane to display the context menu.
 - d. Select **Expand whole Tree** from the context menu.
 - e. Select a settings group in the left pane to see the settings for that group.
5. Click **Close Window** from the **File** menu to close the Connection Properties window.
6. Click **Cancel** to close the Connection Properties dialog box.

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- *Creating advanced configurations* on page 3-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*
 - *BoardChip_name* on page A-9.

3.6.9 Creating advanced configurations

Advanced configurations extend the target visibility and provide more flexibility when debugging multiprocessor development platforms. Typical advanced configurations include:

- Create BCD files containing board/chip definitions that define:
 - custom memory maps
 - memory mapped registers
 - memory mapped peripherals.
- For multiprocessor configurations, you can:
 - create target-specific Advanced_Information blocks to provide a different configuration for each target on your development platform
 - set up a pre-connect sequence to make sure that targets are connected in a specific order
 - specify commands required for cross-triggering.

You create advanced configurations with the Connection Properties window.

Note

You also use the Connection Properties window if you want to hide or show a Debug Configuration.

To display the Connection Properties window:

1. Right-click on the required Debug Configuration to display the context menu.
2. Select **Properties** from the context menu. The Connection Properties dialog box is displayed.
3. Click the **Advanced** button. The Connection Properties window is displayed. Figure 3-9 shows an example:

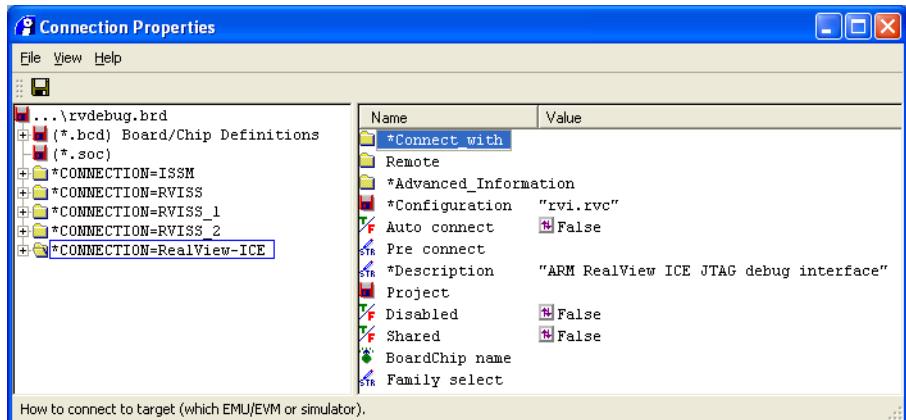


Figure 3-9 Connection Properties window

See also

- *Customizing a Debug Configuration* on page 3-20
- *Displaying the Connection Properties dialog box* on page 3-21
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Creating a target-specific Advanced_Information group* on page 3-23
 - *Configuring the CLI commands for hardware cross-triggering* on page 3-33

- *Configuring a connection sequence for multiple targets* on page 3-36
- Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*
- *Pre_connect* on page A-8
- *Pre_connect* on page A-16
- *Memory mapping Advanced Information settings reference* on page A-20.

3.7 Connecting to a target

Before you can load an image for debugging, you must connect to the required target. The target can be a simulated target or a hardware target.

See also:

- *The Connect to Target window* on page 3-2
- *Making a connection from the Connect to Target window*
- *Making a connection from the Home Page* on page 3-28
- *Connections to RTSMs* on page 3-29
- *Connecting to a Model Library target* on page 3-30
- *Connecting to a Model Process target* on page 3-30
- *Connecting to a SoC Designer target* on page 3-30
- *Effect of connecting to a target on the RealView Debugger GUI* on page 3-32
- *Considerations when connecting to targets* on page 3-33.

3.7.1 Making a connection from the Connect to Target window

To connect to a target:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-10 shows an example:

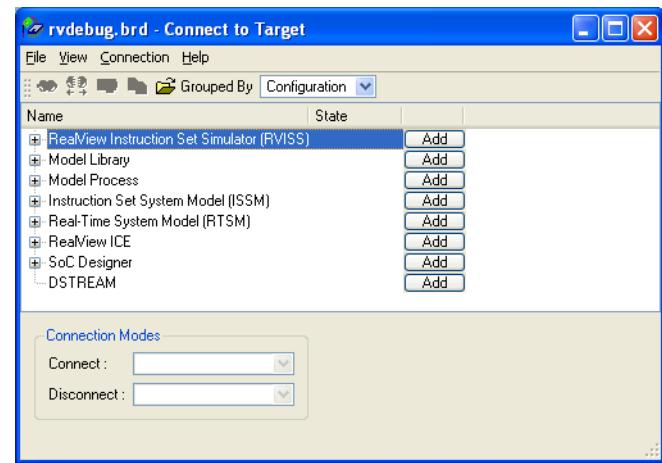


Figure 3-10 Debug interfaces in the Connect to Target window

2. A Debug Interface requires initial configuration before you can connect to any targets on your development platform. However, the RealView Instruction Set Simulator (RVISS) Debug Interface has a default Debug Configuration.
 - If you want to use the default RVISS Debug Configuration, then no additional configuration is required.
 - If you want to connect to a target on any other Debug Interface, or you want to connect to a different RVISS simulated target, then create a new Debug Configuration before you continue with the next step.

————— Note —————

Some ISSMs are provided with RVDS and RVDS Professional.

Some RTSMs are provided with RVDS Professional.

3. Select the required target grouping in the Grouped By field, either:

- **Target**
- **Configuration.**

— Note —

RealView Debugger remembers the chosen grouping when you exit. ARM recommends that you use the **Configuration** grouping if you intend to maintain multiple configurations. For some interfaces, such as Real-Time System Model (RTSM), Model Library, and RealView ICE), the **Target** view imposes unacceptable performance overheads when there are a number of configurations under the interface.

4. Expand the Debug Interface.

5. If you selected the Configuration grouping, expand the Debug Configuration. Otherwise, skip this step.

The list of targets available through the chosen Debug Configuration is expanded, and the connection state of each target is shown (see Figure 3-11).

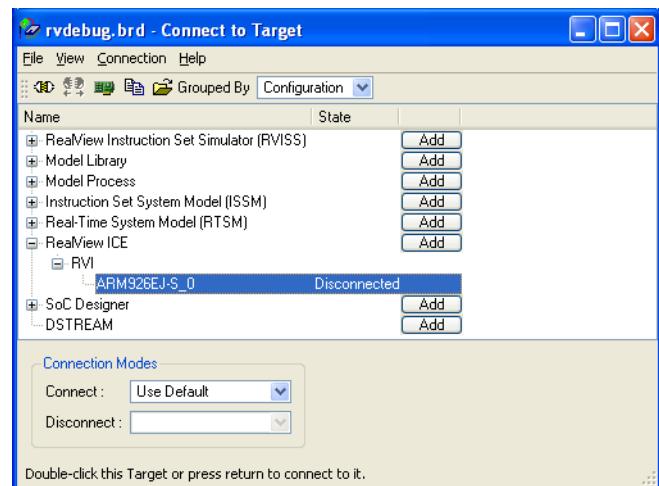


Figure 3-11 Targets in a Debug Configuration

6. Double-click on the target to which you want to connect.

For example, ARM926EJ-S_0 as shown in Figure 3-11.

RealView Debugger connects to the target, and the connection state changes to Connected.

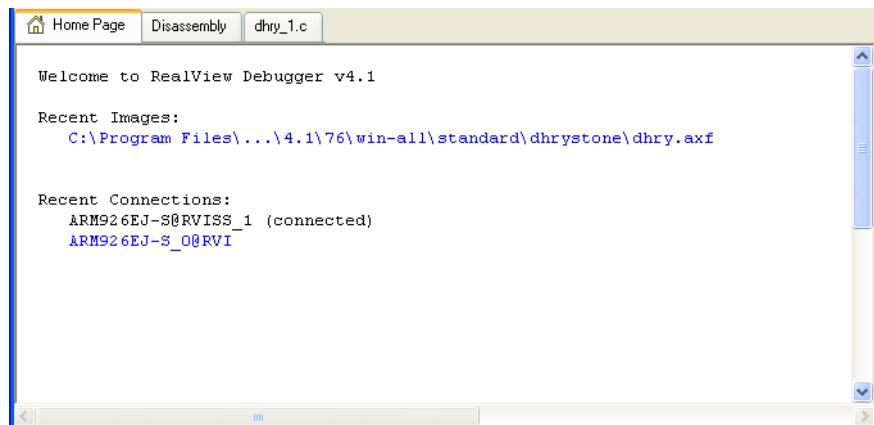
See also

- *About creating a Debug Configuration* on page 3-8.

3.7.2 Making a connection from the Home Page

The **Home Page** lists the last 10 connections you have made. If those connections are still valid, then you can connect to them without having to open the Connect to Target window:

1. Select **Home Page** from the **View** menu to display the **Home Page** in the Code window. Figure 3-12 on page 3-29 shows an example:

**Figure 3-12 Home Page**

The Recent Connections are listed at the bottom of the page.

————— **Note** —————

The Recent Images list is displayed only if you have previously loaded an image.

2. If any connections are listed, click the required connection to connect to the related target. The connection entry changes to black text to show that this connection is established, and the hyperlink is removed until you disconnect.
3. If no connections are listed:
 - a. Click the blue **Connect to Target...** hyperlink. The Connect to Target window is displayed.
 - b. Connect to a target using the Connect to Target window.

See also

- *Making a connection from the Connect to Target window* on page 3-27
- *Loading an image from the Home Page* on page 4-6.

3.7.3 Connections to RTSMs

When you connect to RTSMs, a Real-Time System Model CLCD window is displayed. Figure 3-13 shows an example:

**Figure 3-13 Real-Time System Model CLCD window**

See also

- *Considerations when running applications on RTSMs* on page 8-5
- *RealView Development Suite Real-Time System Model User Guide*.

3.7.4 Connecting to a Model Library target

To connect to a Model Library target:

1. In RealView Debugger, select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-10 on page 3-27 shows an example.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Model Library Debug Interface.
4. Expand the required Model Library Debug Configuration.
5. Double-click on the target to connect.

See also

- *About creating a Debug Configuration* on page 3-8.

3.7.5 Connecting to a Model Process target

To connect to a Model Process target:

1. Start your model running. This can be as standalone executable or within a model simulator.
2. In RealView Debugger, select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-10 on page 3-27 shows an example.
3. Select **Configuration** from the Grouped By drop-down list.
4. Expand the Model Process Debug Interface.
5. If a connection already exists for the model:
 - a. Expand the required Model Process Debug Configuration.
 - b. Double-click on the target to connect.

If a connection does not already exist for the model, you must add a new Debug Configuration.

Note

Be aware that the configuration is keyed to the process ID of the simulation for which the configuration is created. The connection fails if you create a new simulation, even if the new simulation appears to be the same as that already configured.

See also

- *About creating a Debug Configuration* on page 3-8.

3.7.6 Connecting to a SoC Designer target

To connect to a model created with SoC Designer:

1. Open the required model in Carbon SoC Designer Simulator.
2. Start RealView Debugger.
3. In RealView Debugger, select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-10 on page 3-27 shows an example.

4. Select **Configuration** from the Grouped By drop-down list.
5. Expand the SoC Designer Debug Interface.
6. Expand the required SoC Designer Debug Configuration.
SoC Designer connections are named *model*.*target*[*n*] where:
 - model* Is the name of the SoC Designer model.
 - target* Identifies the target processor being modeled.
 - n* A unique number for the connection, starting at zero.
7. Double-click on the target to connect. Figure 3-14 shows example targets for an ARM processor model:

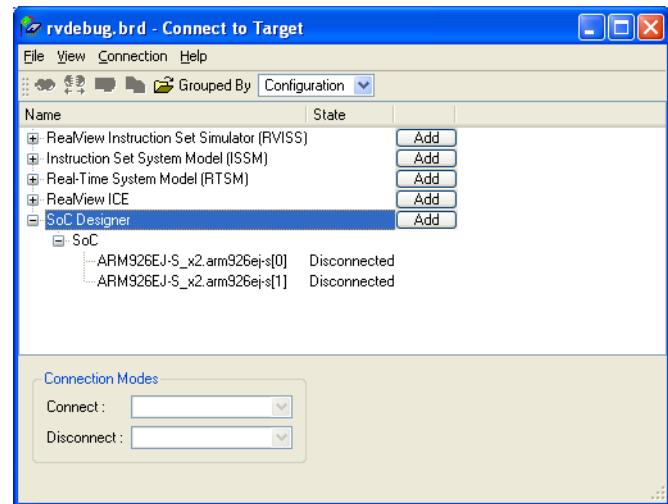


Figure 3-14 SoC Designer model connection in the Connect to Target window

Considerations when debugging SoC Designer models

Be aware of the following when debugging SoC Designer models in RealView Debugger:

- If Carbon SoC Designer Simulator prompts you to load application files when you open a model, then after connecting to the model in RealView Debugger you must load the symbols for the corresponding image (.axf) files that are loaded.
- When you connect to a SoC Designer target, a new Carbon SoC Designer Simulator session is started if:
 - Carbon SoC Designer Simulator is not currently running
 - you started a Carbon SoC Designer Simulator session after starting RealView Debugger.
- Any messages and prompts output by an application running on a SoC Designer model are not displayed in the RealView Debugger Output view. Instead, a message or prompt dialog box is displayed by the Carbon SoC Designer Simulator application. You might have to change focus to Carbon SoC Designer Simulator to see the dialog box.
- If you exit RealView Debugger with the connection to the SoC Designer model still established, then the next time you launch RealView Debugger the connection is re-established if:
 - the same type of model is being debugged (for example, ARM)
 - the connection number matches a previous connection.

This occurs even if you are debugging a different processor model to that previously used. For example, you might have debugged an application on an ARM926E-S™ processor in a previous RealView Debugger session, which used a connection called `mp3_model.arm926e-s[0]`. When you exit RealView Debugger with this connection established, the connection is stored in the RealView Debugger workspace. If you subsequently load a model for an ARM966E-S™ processor, and then launch RealView Debugger, the `mp3_model.arm966e-s[0]` connection is automatically established for the new processor model.

See also

- *Loading symbols only for an image* on page 4-16
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details about the CONFIGURE command
- Carbon SoC Designer Plus, <http://carbondesignsystems.com/Products/SoCDesigner.aspx>.

3.7.7 Effect of connecting to a target on the RealView Debugger GUI

When you connect to a target, the following occurs:

- The target processor, the associated Debug Configuration name, and the associated Debug Interface name are displayed as bold text.



- For targets that support TrustZone® technology, the Code window status bar shows a closed padlock with a red background when the target is in Secure World. This is the initial state after connection.



- If your program changes to Normal World, then the status bar shows an open padlock with a pale blue background.

- The following elements of the RealView Debugger Code window are updated:
 - Title bar, which shows target connection details
 - Toolbar, which includes a color box showing the unique color that RealView Debugger has assigned to the connection.

The menu bar of all windows showing details related to the connection include a color box to identify the connection to which they are associated.

All undocked views include a status bar, which identifies the connection in the parent Code window.

- Status bar, which shows the running state of the target.
- If they are visible, the following views and tabs are updated:
 - **Disassembly** tab in the Code window
 - **Home Page** tab in the Code window (the recent connections list is updated and either an option to load an image or a recent images list appears)
 - Process Control view
 - **Cmd** tab in the Output view
 - Call Stack view
 - for a hardware target, a channel is opened in the Comms Channel view
 - the Diagnostic Log view displays any messages generated by the connection mechanism

- tabs in the Locals view
- Memory view
- Registers view
- Resource Viewer
- Stack view.

See also

- *Overview of RealView Debugger windows and views* on page 1-2.

3.7.8 Considerations when connecting to targets

Be aware of the following when you are connecting to targets:

- If the chosen Debug Interface does not have any Debug Configurations, you must create a new Debug Configuration to identify the targets on your development platform, and then connect.
- RealView Debugger connects to the specified target using a default connection mode. However, you can specify the connection mode to use.
- When connected to a CoreSight component, and that connection is the current connection, only the Registers and Memory views are available in the RealView Debugger Code window.
- The state of the Connect to Target window is saved when you exit RealView Debugger. Therefore, the next time you start RealView Debugger in GUI mode, the Connect to Target window is displayed in the same state. That is:
 - the last used grouping is displayed
 - connections are re-established
 - Debug Interfaces and Debug Configurations that have connected targets are expanded.
- When you first connect to an ARM architecture-based debug target, RealView Debugger might display a warning message in the **Cmd** tab:
Warning: No stack/heap or top_of_memory defined - setting top_of_memory to 0x00020000.
 You can change the top of memory if required:
 - for the current debugging session only
 - on a more permanent basis.
- You can make a connection to a target processor running an OS on your development platform that supports threads.
- If you have configured RVISS to use a map file, the memory map is displayed in the **Mapfile** tab of the Registers view.

See also

- *Setting top of memory for the current debugging session* on page 3-38
- *Connecting to a target using different modes* on page 3-43
- *Troubleshooting target connection problems* on page 3-60

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Setting top of memory* on page 4-24
- the following in the *RealView Debugger RTOS Guide*:
 - Chapter 1 *OS Support in RealView Debugger*
- *RealView ARMulator ISS User Guide*.

3.8 Showing the trace components in the Connect to Target window

To configure individual trace components, the trace components must be visible in the Connect to Target window. If your development platform has trace components, they are hidden by default.

To show the trace components in the Connect to Target window, select **Show NonCores** from the **View** menu.

See also:

- *Viewing information about the target topology* on page 3-36.

3.9 Viewing information about the target topology

If your development platform supports trace components, then you can view information about the target topology.

Note

This feature is supported only for connections through the RealView ICE Debug Interface, and only when using a DSTREAM unit or a RealView ICE unit with v3.1 firmware or later.

To view the target topology:

1. Before you can view the topology for your development platform, you must set up the associations for the target processors and other components. To do this:
 - a. Create a text file containing the device associations for your development platform.
 - b. Display the RVConfig utility for your RealView ICE Debug Configuration.
 - c. Assign the associations file to your Debug Interface configuration.

Example 3-1 shows the associations file entries for the topology shown in Figure 3-15 on page 3-37:

Example 3-1 Trace device associations

```
Name=ARMCS-DP;Type=ARMCS-DP;
Name=Cortex-R4_1;Type=Cortex-R4;ETM=CSETM_1;
Name=CSETM_1;Type=ETM;TraceOutput0=CSTPIU;TraceOutput1=CSETB;Core=Cortex-R4_1;
Name=CSETB;Type=ETB;Port0=CSETM_1;
Name=CSTPIU;Type=TPIU;Port0=CSETM_1;
```

2. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
3. Select **Show NonCores** from the **View** menu to display the non-processor devices.

Note

Non-processor devices are hidden by default.

4. Select **Configuration** from the Grouped By field.
5. Expand the RealView ICE Debug Interface.
6. Expand the Debug Configuration for your development platform. An example of a development platform with CoreSight components is shown in Figure 3-15 on page 3-37:

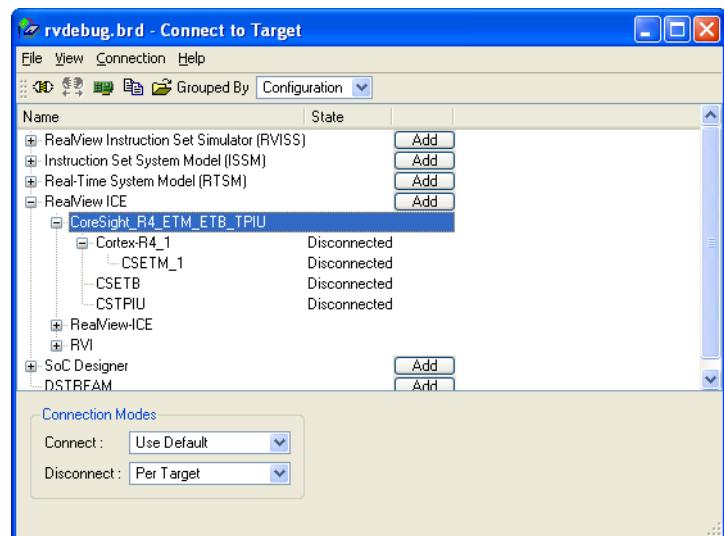


Figure 3-15 Targets on a CoreSight development platform

7. Expand the processor node to show the ETM associated with the processor.
8. Right-click on the component for which you want to view the topology. For the example shown in Figure 3-15, right-click on CSETM_1.
9. Select **Topology** from the context menu. The Topology dialog box is displayed. Figure 3-16 shows an example:

————— Note —————

The **Topology** menu option is enabled only when the associated Debug Interface configuration has an associated device topology.



Figure 3-16 Topology dialog box

See also:

- *Connections to trace components* on page 3-5
- *Showing the trace components in the Connect to Target window* on page 3-35
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities* for details on how to create an associations file, and how to assign the associations to your Debug Interface configuration
- <http://www.arm.com/support/faqdev/18377.html> for more information on Association Files.

3.10 Setting top_of_memory for the current debugging session

You can set the `top_of_memory` on a temporary basis, that is for as long as the current connection is established, using the `@top_of_memory` symbol.

Note

Top of memory is valid only for targets connected through DSTREAM or RealView ICE.

To set `top_of_memory` for a connection in the current debugging session:

1. Select **Registers** from the **View** menu to display the Registers view. Figure 3-17 shows an example:



Figure 3-17 Top of Memory in Debug tab of Registers view

2. Locate the `Top of Memory` setting.
3. Click on the value for the `Top of Memory` setting.
4. Replace the current value with the required value for the top of memory.
5. To check the value of `top_of_memory`, enter the CLI command:
`printvalue @top_of_memory`

See also:

- *Configuring top of memory* on page 3-23
- *Setting top_of_memory and stack values*
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

3.10.1 Setting top_of_memory and stack values

If defined, the `top_of_memory` variable specifies the highest address in memory that the C library can use for stack space. By default, a semihosting call returns `top_of_memory`. Stack and base are then set to be immediately below `top_of_memory`.

RealView Debugger uses default settings for stack and base that are target-dependent. The default setting for `top_of_memory` depends on the target. The default is `0x80000` when connecting to an ARM architecture-based processor through DSTREAM or RealView ICE.

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

Warning: No stack/heap or top_of_memory defined - setting top_of_memory to 0x00020000.

To avoid this message for connections to RVISS targets, permanently set top_of_memory using the Connection Properties window. Configure this for your debug target so that it is used whenever you connect with RealView Debugger.

— **Note** —

Top of memory is valid only for targets connected through DSTREAM or RealView ICE.

See also:

- *Setting top of memory for the current debugging session* on page 3-38
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

3.11 Viewing connection details

As you establish a new connection, the connection details are shown in:

- the Code window title bar
- the Output view.

See also:

- *Connection details in the Code window title bar*
- *Connection details in the Output view* on page 3-41.

3.11.1 Connection details in the Code window title bar

When you connect to a target, your title bar looks like the one shown in Figure 3-18.

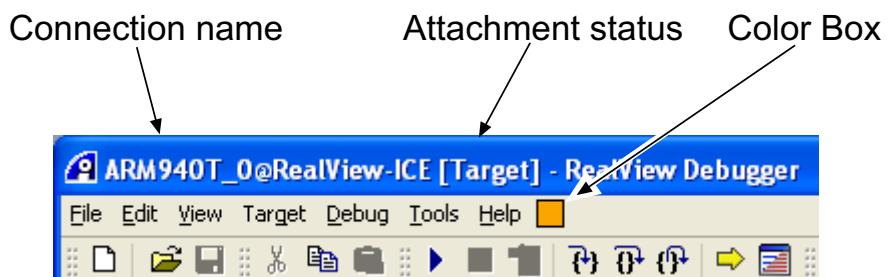


Figure 3-18 Connection information in the Code window title bar

This shows:

@ARM940T_0@RealView-ICE

The connection name, incorporating the processor name, the connection number, and the Debug Configuration name. In this example, the connection is to an ARM940T™ processor using the RealView-ICE Debug Configuration through a RealView ICE Debug Interface.

[Target] The attachment of the window to the connection currently displayed in the Code window. By default, a Code window is not attached to a connection and the attachment status is not displayed.

When you attach a Code window to the connection shown in that Code window, the status [Target] is added after the connection name.

————— Note —————

If you are debugging OS-aware images and the title bar contains no attachment details, then this window is attached to the current thread.

The color box is a visual cue for each connection you make. A different color is allocated for each active connection, and the color box is displayed in each window, or floating view, that is displaying information for that connection.

See also

- *Attaching a Code window to a connection* on page 7-10
- the following in the *RealView Debugger RTOS Guide*:
 - *Attaching and unattaching windows* on page 4-2.

3.11.2 Connection details in the Output view

As you connect to a target, the messages are displayed in the **Cmd** tab of the Output view. The following example shows the messages that might be displayed when connecting to an ARM966E-S through RealView ICE.

```
> connect @ARM966E-S_0@RVI
Advanced_info searched in: Local Advanced_info, BOARD=CM966ES, BOARD=CP
Using Advanced info based on Processor 'ARM'
Using Advanced info based on 'Default' or 'All' Mode: Little Endian
ARM RealView ICE
Base H/W: V1 Rev C-01
TurboTAP Rev: 1.71
Firmware: 1.4.0, Build 450
Copyright ARM Limited 2002,2003,2004,2005
Attached to stopped device
Warning: No stack/heap or top_of_memory defined - setting top_of_memory to 0x00020000.
```

3.12 Connecting to a target on startup

You can automatically connect to a target when you start RealView Debugger.

To connect to a target when you start RealView Debugger, enter the `rvdebug` command with the `--init` argument:

```
rvdebug --init=@target@DebugConfiguration
```

For example:

- to connect to an ARM940T target through a RealView ICE Debug Interface using the RealView-ICE Debug Configuration, enter:

```
rvdebug --init=@ARM940T_0@RealView-ICE
```

- to connect to a Cortex-A8 simulated target in the ISSM Debug Configuration, enter:

```
rvdebug --init=@ARM_Cortex-A8_0@ISSM
```

See also:

- *Starting RealView Debugger from the command line* on page 2-2
- *Storing connections when exiting RealView Debugger* on page 3-59.

3.13 Connecting to a target using different modes

You can control the way RealView Debugger connects to a processor. For example, you might want to connect to a target that is already running an application from a previous session. This is useful when debugging multiprocessor platforms or multithreaded applications, but can also be used when debugging a single processor platform, for example using RealMonitor.

See also:

- *The connect modes*
- *Setting connect mode for the current debugging session* on page 3-44
- *Displaying the connect mode in the Connect to Target window* on page 3-44
- *Setting connect mode permanently* on page 3-44
- *Considerations for DSTREAM or RealView ICE* on page 3-44
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43 for more details on using RealMonitor.

3.13.1 The connect modes

The connect modes available depend on your target:

Use default This is the default option for all targets. When selected, the connect mode used depends on whether or not you have changed the Connect_mode setting in the connection properties for the connection:

- If you have not set Connect_mode in the connection properties, then RealView Debugger connects using the default mode, **No Reset and Stop**.
- If you have set Connect_mode in the connection properties, then RealView Debugger connects using the mode you have specified in the setting.

No Reset and Stop

Does not submit a processor reset but explicitly halts any process currently running by issuing a Stop command.

————— Note —————

This is the only mode available when connecting to a simulated target through the RVISS Debug Interface.

No Reset and No Stop

Does not submit a processor reset or attempt to halt any process currently running.

Reset and Stop

Submits a processor reset and explicitly halts any process currently running by issuing a Stop command.

Reset and No Stop

Submits a processor reset but does not attempt to halt any process currently running.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Specifying connect and disconnect mode* on page 3-19.

3.13.2 Setting connect mode for the current debugging session

To connect to a target using a specific connect mode:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-1 on page 3-2 shows an example.
2. Select the required grouping from the Grouped By field. For example, select **Configuration**.
3. Expand the required Debug Interface.
4. Expand the required Debug Configuration to see the list of targets available through the that Debug Configuration.
5. Select the target to which you want to connect.
6. If required, choose the required Connect Mode in the Connection Mode group.
7. Double-click on the target to connect.

Note

When you set the connect mode in this way, the mode persists for the target only during the current debugging session. This enables you to temporarily override the `Connect_mode` setting in the connection properties.

3.13.3 Displaying the connect mode in the Connect to Target window

To display the connect mode for each target in the Connect to Target window, select **Connect Mode** from the **View** menu of the Connect to Target window. The Connect Mode column is added to the window.

3.13.4 Setting connect mode permanently

To permanently set the connect mode for a target, configure the connect mode using the `Connect_mode` setting in the connection properties.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Specifying connect and disconnect mode* on page 3-19.

3.13.5 Considerations for DSTREAM or RealView ICE

For connections through DSTREAM or RealView ICE, what happens when a processor comes out of reset depends on the combination of:

- the RealView Debugger connection mode
- the Default Post Reset State setting in the RVConfig utility.

Table 3-3 summarizes the actions.

Table 3-3 Action performed after reset

Connection mode	Action performed after reset
Reset and Stop	As specified by the RVConfig Default Post Reset State setting.
Reset and No Stop	As specified by the RVConfig Default Post Reset State setting.
No Reset and Stop	As specified by the RealView Debugger connect_mode setting or Connect Mode on the Connect to Target window.
No Reset and No Stop	As specified by the RealView Debugger connect_mode setting or Connect Mode on the Connect to Target window.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *About customizing a DSTREAM or RealView ICE Debug Interface configuration* on page 2-3 for details of the Default Post Reset State setting
 - *Specifying connect and disconnect mode* on page 3-19.

3.14 Connecting to multiple targets

If you are developing a multiprocessor application, then it is essential to be able to debug all parts of your development platform concurrently. Therefore, you must be able to connect to all the processors so that you can debug the images running on each processor.

This task assumes that you are familiar with the information in:

- *About target connection* on page 3-2
- *Connecting to a target* on page 3-27.

Also, you must have configured your Debug Interface to recognize all the targets on your development platform. The targets can be ARM architecture-based processors or CoreSight components.

You can connect to multiple targets in the following ways:

- Connect to each target individually in the same way that you connect to a single target.
- If all your targets are accessible through a single Debug Configuration, you can connect to all the targets in a single operation.

With multiple connections established, you can:

- Open multiple Code windows, closely associate each target connection to a different Code window, and load an image to each target ready for debugging.
- Synchronize the processors on each connection.
- Set up cross-triggering, so that one processor can control the actions of the other processors.

See also:

- *Consideration when connecting to multiple targets*
- *Changing the current target connection* on page 3-50
- *Attaching a Code window to a connection* on page 7-10
- *Synchronizing multiple processors* on page 7-13
- *Connecting to a target* on page 3-27
- *Setting up software cross-triggering* on page 7-17
- *Connecting to all targets for a specific Debug Configuration* on page 3-48.

3.14.1 Consideration when connecting to multiple targets

As you connect to each target:

- The connection details for the last target connection you established are displayed in any unattached Code windows.
- The connection that has its details visible in an unattached Code window is referred to as the *current connection*. The current connection is usually the last connection that you established. However, you can make any existing connection the current connection.
- To load an image or binary to a connected target, the connection must be the one that has its details visible in the Code window that you are using.
- RealView Debugger maintains a list of the existing connections in the *active connections* list. This list is updated as you connect to or disconnect from a target.

- The active connections list shows which connection is the current connection by placing an asterisk in front of that connection. You can use this list to change the current connection.
- By default, all settings in a Debug Configuration are applied to all targets. However, you can create named Advanced_Information block groups for specific connections, processors, or processor families. For example, you might want to define settings that are applied to an ARM926EJ-S processor. Therefore, if you create an Advanced_Information block group called ARM926EJ-S, then any settings in this group are used only for ARM926EJ-S processors.
- The targets can be connected in a particular order by specifying a connection sequence for the Debug Configuration. For example, target A might require that target B is connected first. Therefore, when you connect to the target A, RealView Debugger first connects to target B, then connects to target A.

See also

- *Changing the current target connection* on page 3-50
- *Attaching a Code window to a connection* on page 7-10
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Creating a target-specific Advanced_Information group* on page 3-23
 - *Configuring a connection sequence for multiple targets* on page 3-36.

3.15 Connecting to all targets for a specific Debug Configuration

If you have a multiprocessor development platform, then you can connect to all configured targets for a selected Debug Configuration in a single operation.

— Note —

You must select the **Configuration** grouping to use this feature.

To connect to all targets for a specific Debug Configuration:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-1 on page 3-2 shows an example.
2. Select **Configuration** from the Grouped by list.
3. Expand the required Debug Interface to list the associated Debug Configurations.
4. Expand the required Debug Configuration to list the associated targets.

— Note —

If the Debug Configuration is not correctly configured, then RealView Debugger displays a Target Connection Error dialog box.

5. Right-click on the required Debug Configuration name to display the context menu.
6. Select **Connect** from the context menu. RealView Debugger connects to all the targets in the chosen Debug Configuration.

See also:

- *Considerations when connecting to all targets for a specific Debug Configuration*
- *Failing to make a connection* on page 3-60.

3.15.1 Considerations when connecting to all targets for a specific Debug Configuration

By default, all targets for a specific Debug Configuration are connected in the order listed in that configuration. However, you can change the connection sequence if, for example, a target requires one or more other targets be connected first. You define a target connection sequence using the **Pre_connect** setting in the Connection Properties:

- If no targets are specified for the **Pre_connect** setting, then the targets are connected in the order listed in the Debug Configuration.
- If targets are specified for a **Pre_connect** setting in an **Advanced_Information** block named **Default**, then the targets are connected in the order listed for the setting.
- If targets are specified for the **Pre_connect** setting in a target-specific **Advanced_Information** block, then:
 - the targets are connected in the order listed for the setting
 - the target that matches the name of the **Advanced_Information** block is then connected.

Considerations for development platforms that include trace components

If your development platform includes trace components, you might want to connect any target processors after connecting to the trace components. To do this, you can set up a connection sequence for the Debug Configuration. This ensures that the connection to a target processor is always set up as the current connection. Alternatively, you can cycle through the connections after they have been established.

To perform tracing on a target processor, the details of the connection to that processor must be shown in a Code window. Typically, this is the current connection. To make sure that the connection details for a processor are always visible in a Code window, even when you cycle through connections, then attach the Code window to that connection.

See also

- *Viewing information about the target topology* on page 3-36
- *Connecting to multiple targets* on page 3-46
- *Changing the current target connection* on page 3-50
- the following in the *RealView Debugger Target Configuration Guide*:
 - *The Debug Configuration Advanced_Information block* on page 3-8
 - *Creating a target-specific Advanced_Information group* on page 3-23
 - *Configuring a connection sequence for multiple targets* on page 3-36.

3.16 Changing the current target connection

If you have multiple target connections, the last target connection you made is the current target connection by default. However, you can choose a different target connection to be the current connection.

See also:

- *What is the current connection?*
- *Cycling through the active connection list*
- *Selecting a connection from the active connection list* on page 3-51
- *Considerations when changing the current connection* on page 3-51.

3.16.1 What is the current connection?

The term *current connection* is used to denote the selection mechanism that RealView Debugger uses to decide what to display and to provide the scope of CLI commands. The current connection is usually the last connection that you make.

Connections are queued in the order in which they are made. As a new connection is added to the list of available connections, it becomes the current connection. If you disconnect the current connection, the next available connection in the list automatically becomes current.

You can change the current connection yourself so that the Code window displays details for a different target.

A single RealView Debugger Code window always displays information relating to the current connection unless you attach a specific connection to that window.

See also

- *Attaching a Code window to a connection* on page 7-10.

3.16.2 Cycling through the active connection list

You can change the current target connection by cycling through the active connection list.



To cycle through the active connections, click **Cycle Connections** in the Code window toolbar until the details for the connection you require are displayed in the Code window. This connection is now the current connection.

3.16.3 Selecting a connection from the active connection list

To select a connection from the active connection list:

1. Either:

- Select **Connections** from the **Target** menu of the Code window.
- Click the drop-down arrow on the **Cycle Connections** button.



The active connection list is displayed, shown in Figure 3-19.

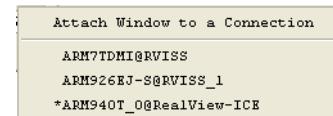


Figure 3-19 Active connection list

The list shows the active connections in the order in which they are established.

The current connection is marked with an asterisk, for example:

*ARM940T_0@RealView-ICE

This is usually the last connection established.

2. Select the required connection from the list.

3.16.4 Considerations when changing the current connection

Be aware of the following:

- Changing the current connection immediately updates all unattached Code windows. The new connection is shown in the title bar and the color box changes color to indicate the new connection.
- If you change the current connection from an attached Code window, only the context of unattached windows changes to that of the current connection. The attached Code window still shows the context of the connection to which it is attached. Unattached Code windows always display the current connection.

3.17 Disconnecting from a target

There are several ways to disconnect when working with a target. Choosing the most appropriate method depends on:

- the number and attachment of Code windows
- which window has the focus when the disconnection option is used
- the state of the currently connected processor, and process if running
- the required state of the processor, or process, following disconnection.

See also:

- *Disconnecting from any target*
- *Disconnecting from the target shown in the Code window*
- *Disconnect behavior in RealView Debugger*
- *Considerations when disconnecting from targets* on page 3-53.

3.17.1 Disconnecting from any target

To disconnect from a target:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window. If any targets are connected then the associated Debug Interface name is shown in bold text.
2. Expand the required Debug Interface.
3. In the Configuration grouping expand the required Debug Configuration.
4. Double-click on the target to be disconnected.

The target is disconnected, and the connection state changes to **Not Connected**.

See also

- *Disconnect behavior in RealView Debugger.*

3.17.2 Disconnecting from the target shown in the Code window

To disconnect from the connection that is visible in the Code window that you are currently using, either:



- Click **Disconnect** on the Connect toolbar.
- Select **Disconnect** from the **Target** menu.

This immediately terminates the connection that is shown in the Code window.

3.17.3 Disconnect behavior in RealView Debugger

Code windows are not closed when disconnecting. However, the contents might change depending on whether or not you have other connections:

- if you have other connections, and you disconnect the current connection, then the next connection in the list is set to the current connection, and the process details for that connection are displayed
- if you have other connections, and you disconnect a connection that is not the current connection, then the Code window does not change

- if there are no other connections, then all connection and process details are removed.

Additional behavior depends on the update options you set for the window and the disconnect state of the target processor:

- any images that are loaded on the disconnected target are unloaded
- the associated source files close, if they have been automatically opened by RealView Debugger, and you have not edited them
- entries displayed in the various views are cleared.

3.17.4 Considerations when disconnecting from targets

When you disconnect from a target, RealView Debugger uses a default disconnect mode. However, you can change the disconnect mode if required.

— Note —

All processor exceptions (that is, vector catch breakpoints) are removed from the target on disconnection, irrespective of the disconnect mode.

Disconnecting with an image loaded

If you disconnect with an image loaded, this removes debug information from RealView Debugger and so clears view contents from your Code window. You can reload the image if you reconnect.

See also

- *Disconnecting from a target using different modes* on page 3-54
- *Specifying processor exceptions (global breakpoints)* on page 11-65
- *Chapter 4 Loading Images and binaries*.

3.18 Disconnecting from a target using different modes

You can control the way RealView Debugger disconnects from a processor. This is useful when debugging multiprocessor systems or multithreaded applications, but can also be used when debugging a single processor target system, for example using RealMonitor.

See also:

- *The disconnect modes*
- *Setting disconnect mode for the current debugging session*
- *Displaying the disconnect mode in the Connect to Target window* on page 3-55
- *Setting disconnect mode permanently* on page 3-55
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43 for more details on using RealMonitor.

3.18.1 The disconnect modes

The disconnect modes available depend on your target, and can be one of the following:

As-is without Debug

Leaves the target in its current state, whether stopped or running, and removes any debugging controls such as software breakpoints. This is the default.

If this leaves the processor running, any defined breakpoints are disabled. This means the program does not enter debug state after the debugger has disconnected.

As-is with Debug

Leaves the target in its current state, whether stopped or running, and maintains any debugging controls such as software breakpoints.

If this leaves the processor running, any defined breakpoints are still active. This means the program might enter debug state after the debugger has disconnected, depending on the code paths the program takes.

Note

All processor exceptions (that is, vector catch breakpoints) are removed from the target on disconnection, irrespective of the disconnect mode.

See also

- *Specifying processor exceptions (global breakpoints)* on page 11-65.

3.18.2 Setting disconnect mode for the current debugging session

To disconnect from a target using a specific disconnect mode:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window. If any targets are connected then the associated Debug Interface name is shown in bold text.
2. Display the connected target:
 - In the **Target** grouping, expand the required Debug Interface.

- In the **Configuration** grouping, expand the required Debug Interface and Debug Configuration.
3. Select the target to be disconnected.
 4. In the Connection Mode group choose the required Disconnect Mode.
 5. Double-click on the target to disconnect.
- The target is disconnected, and the connection state changes to Not Connected.

Note

When you set the disconnect mode in this way, the mode persists for the target only during the current debugging session. This enables you to temporarily override the `Disconnect_mode` setting in the connection properties.

See also

- *Disconnect behavior in RealView Debugger* on page 3-52.

3.18.3 Displaying the disconnect mode in the Connect to Target window

To display the disconnect mode for each target in the Connect to Target window, select **Disconnect Mode** from the **View** menu of the Connect to Target window. The Disconnect Mode column is added to the window.

3.18.4 Setting disconnect mode permanently

To permanently set the disconnect mode for a target, configure the disconnect mode using the `Disconnect_mode` setting in the connection properties.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Specifying connect and disconnect mode* on page 3-19.

3.19 Disconnecting from multiple targets

You can disconnect from multiple targets either individually or in a single operation.

See also:

- *Methods for disconnecting from multiple targets*
- *Considerations when disconnecting from multiple targets*
- *Effect on Code windows when disconnecting* on page 3-57.

3.19.1 Methods for disconnecting from multiple targets

You can disconnect from multiple targets in the following ways:

- Disconnect from each target individually in the same way that you disconnect from a single target.
- If all your targets are accessible through a single Debug Configuration, you can disconnect from all the targets in a single operation.
- To disconnect from all targets in a single operation for all Debug Configurations, enter the CLI command:
`disconnect,all`

At any point in your debugging session, you can disconnect from a target using the Connect to Target window.

See also

- *Disconnecting from a target* on page 3-52
- *Disconnecting from all targets for a specific Debug Configuration* on page 3-58
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details about the DISCONNECT command.

3.19.2 Considerations when disconnecting from multiple targets

If the current connection is terminated, this has the following results:

- The active connection list is updated.
- Any windows attached to the current connection are detached.
- The next connection in the active connections list becomes the current connection.
- Title bars and color boxes for all unattached windows are updated to reflect the new current connection.
- Any window that is attached to another connection in the active connections list is only affected if that connection becomes the new current connection. In this case, the title bars are updated to show that the window is attached to the current connection.

If the terminated connection is not the current connection, this has the following results:

- The chosen connection is terminated.
- All active connections lists are updated.
- Any windows attached to the connection chosen for termination are unattached.

- Title bars and color boxes for all newly-unattached windows are updated to reflect the current connection.

If the connection chosen for termination is the only remaining connection, then the **Cycle Connections** button is disabled.

— Note —

All processor exceptions (that is, vector catch breakpoints) are removed from the targets on disconnection, irrespective of the disconnect mode.

3.19.3 Effect on Code windows when disconnecting

Code windows are not closed when you disconnect, but the contents of the windows might change. If you disconnect from a target that is attached to a Code window, then the details in that Code window change to those for the current connection.

See also

- *Considerations when disconnecting from targets* on page 3-53.

3.20 Disconnecting from all targets for a specific Debug Configuration

If all your connections are to targets on a single Debug Configuration, then you can disconnect from all the targets in a single operation.

— Note —

You must select the **Configuration** grouping to use this feature.

You can disconnect from all connected targets for a selected Debug Configuration in a single operation. To do this:

1. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window. Figure 3-1 on page 3-2 shows an example.
2. Select **Configuration** from the Grouped by list.
3. Expand the required Debug Interface to list the associated Debug Configurations.
4. Right-click on the required Debug Configuration name to display the context menu.
5. Select **Disconnect** from the context menu. The targets are disconnected in the order shown in the Debug Configuration.

— Note —

The **Disconnect** option is enabled only when you select a Debug Configuration that has connected targets.

3.21 Storing connections when exiting RealView Debugger

You do not have to close all your connections before you exit RealView Debugger, because RealView Debugger closes all connections on exit.

The details of the connection are stored in the current workspace.

— Note —

RealView Debugger disconnects using the disconnect mode defined in the connection configuration details.

When you next start RealView Debugger, the stored connections are re-established. However, if any connection cannot be re-established, RealView Debugger displays an error dialog box.

See also:

- Chapter 17 *Configuring Workspace Settings*
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 3 *Customizing a Debug Configuration*.

3.22 Troubleshooting target connection problems

The following sections helps you to identify and fix connection problems you might encounter when connecting to targets:

- *Failing to make a connection*
- *Problems connecting to targets in a system containing trace components* on page 3-61
- *Error P1001E (Parser): Specified target not in list of available targets* on page 3-61
- *Simulator Debug Configuration error states* on page 3-62
- *DSTREAM or RealView ICE Debug Configuration error states* on page 3-62
- *Problems with configuration files* on page 3-63
- *Kill all other connections error with DSTREAM or RealView ICE* on page 3-63
- *Examining details in the Diagnostic Log view* on page 3-63
- *Other problems* on page 3-64.

3.22.1 Failing to make a connection

RealView Debugger might fail to connect to your chosen debug target because of the following reasons:

- you do not have a valid license to use the debug target (RealView Debugger displays a message if this is the case)
- the debug target is not installed or the connection is disabled
- the target hardware is in use by another user
- the connection has been left open by software that exited incorrectly
- the target has not been configured, or a configuration file cannot be located
- the target hardware is not powered up ready for use
- the target is on a scan chain that has been claimed for use by something else
- the target hardware is not connected.

If a Debug Configuration is not correctly configured, RealView Debugger displays the Target Connection Error dialog box shown in Figure 3-20.

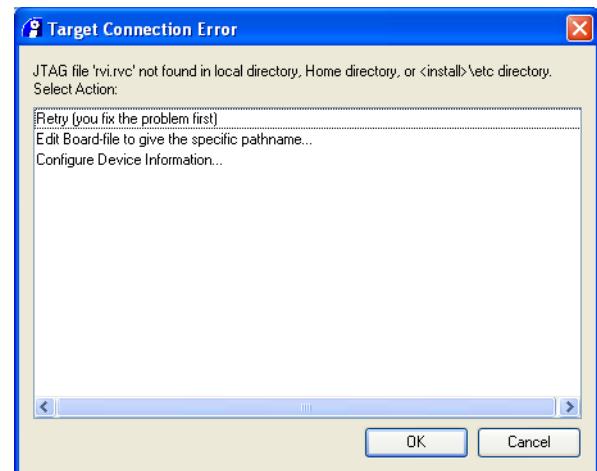


Figure 3-20 Failing to make a connection

The message displayed at the top of the selection box indicates the type of problem encountered by RealView Debugger. The message displayed, and the options available, depends on the Debug Interface you are using:

- If you have identified the cause of the failure and corrected it, for example you have connected a board or switched on power, then:
 1. Select **Retry (you fix the problem first)**
 2. Click **OK** to connect.
- To close the list selection box and display the Connection Properties window where you can edit your target configuration details:
 1. Select **Edit Board-file to give the specific pathname...**
 2. Click **OK**. The Connection Properties dialog box is displayed.
 3. Click **Advanced**. The Connection Properties window is displayed.
 4. Correct the configuration settings.
 5. Select **Save and Close** from the **File** menu to save your changes and close the Connection Properties window.
 6. Click **OK** to close the Connection Properties dialog box.
- To display the configuration dialog box for the target:
 1. Select **Configure Device Information...**
 2. Click **OK**.

You must configure certain targets before you can make your first connection, for example targets connected through DSTREAM or RealView ICE.
- To close the message box and abandon the connection, click **Cancel**.

See also

- *About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 3 *Customizing a Debug Configuration*.

3.22.2 Problems connecting to targets in a system containing trace components

If you have problems connecting to targets in a system containing trace components, then check the following:

- Make sure you have assigned an associations file to the Debug Interface configuration.
- Make sure the associations file accurately describes the system configuration.

See also

- *Connections to trace components* on page 3-5
- *Viewing information about the target topology* on page 3-36
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

3.22.3 Error P1001E (Parser): Specified target not in list of available targets

You attempted to connect to a target by:

- clicking on a recent connection in the **Home Page**

- selecting a recent connection from the **Recent Connections** menu on the Code window **Target** menu
- entering a CONNECT command, either directly or within a script.

In these cases, the error message Error P1001E (Parser): Specified target not in list of available targets is displayed for the following reasons:

- You have disabled the Debug Configuration that defines the target connection. Enable the Debug Configuration again.
- You have renamed or deleted the Debug Configuration that defines the target connection. If you do not have another Debug Configuration for the development platform, create a new Debug Configuration.

See also

- About creating a Debug Configuration* on page 3-8
- Deleting a Debug Configuration* on page 3-19
- the following in the *RealView Debugger Target Configuration Guide*:
 - Hiding a Debug Configuration* on page 3-18.

3.22.4 Simulator Debug Configuration error states

The possible error state for RVISS, ISSM, RTSM, and SoC Designer Debug Configurations is:

Not Configured

This state is displayed when:

- the associated configuration file is missing
- no configuration file is specified for the Debug Configuration
- the configuration file has become corrupted
- a reference to a board/chip definition in the Connection Properties is invalid.

In the first two cases, the Target Connection Error dialog box is displayed.

Additional error or warning messages might be displayed in the Output view.

See also

- Failing to make a connection* on page 3-60
- About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - Troubleshooting Debug Configurations* on page 3-65
 - Debug Configuration generic groups and settings* on page A-6.

3.22.5 DSTREAM or RealView ICE Debug Configuration error states

The possible error states for a DSTREAM or RealView ICE Debug Configuration are:

Hardware Unavailable

This state is displayed when:

- the network or USB cable of the DSTREAM or RealView ICE interface unit is disconnected
- the DSTREAM or RealView ICE interface unit is powered off.

Not Configured

This state is displayed when:

- the associated configuration file is missing
- no configuration file is specified for the Debug Configuration
- the configuration file has become corrupted.

In the first two cases, the Target Connection Error dialog box is displayed.

General Error

This state is displayed when:

- you attempt to access a target that is powered off
- a reference to a board/chip definition in the Connection Properties is invalid.

Additional error or warning messages might be displayed in the Output view and the Diagnostic Log view.

See also

- *Failing to make a connection* on page 3-60
- *About creating a Debug Configuration* on page 3-8
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Troubleshooting Debug Configurations* on page 3-65
 - *Debug Configuration generic groups and settings* on page A-6.

3.22.6 Problems with configuration files

If your working versions of configuration files are accidentally deleted, or become corrupted, RealView Debugger is unable to determine your target configuration.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Restoring the default connections and configurations* on page 3-55.

3.22.7 Kill all other connections error with DSTREAM or RealView ICE

If you are using a DSTREAM or RealView ICE unit, and see a message asking you to Kill all other connections..., ensure that the required connection is available before terminating other connections.

See also

- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

3.22.8 Examining details in the Diagnostic Log view

The Diagnostic Log view provides a detailed log of messages that are generated when you connect to a target. This view might help to track down any problems with target connections.

The Diagnostic Log view also enables you to apply filters to remove any unwanted messages from the display. Be aware that the filters persist. Therefore, if the messages do not appear to be helpful, make sure you remove any filtering you have applied.

Displaying the Diagnostic Log view

To display the Diagnostic Log view, select **Diagnostic Log** from the Code window **View** menu.

Filtering messages in the Diagnostic Log view

You can filter messages by component or by severity.

To filter messages for a specific component or severity:

1. Right-click on a message with the required component or severity.
2. Select the required filter option from the context menu. Table 3-4 describes the context menu options.

Table 3-4 Menu options for filtering messages in the Diagnostic Log view

Context menu option	Description
Include → Component	Displays only those messages that have the same component as the chosen message.
Include → Severity	Displays only those messages that have the same severity as the chosen message.
Exclude → Component	Hides only those messages that have the same component as the chosen message.
Exclude → Severity	Hides only those messages that have the same severity as the chosen message.
Reset Filter	Clears all filters that you have applied, so that messages for all components and severities are displayed.

Saving Diagnostic Log messages to a CSV file

To save the displayed messages in the Diagnostic Log view to a *Comma-Separated Value* (CSV) file:

1. Apply any message filtering that you require.
2. Right-click on a message.
3. Select **Save...** from the context menu. The Save dialog box is displayed.
4. Enter an appropriate filename.
5. Click **Save**. The messages are saved to a .csv file.

See also

- *Diagnostic Log view* on page 1-14.

3.22.9 Other problems

The Target Connection Error dialog box, shown in Figure 3-20 on page 3-60, might include the entry:

Display list of possible problems...

RealView Debugger might display this option if there are known problems with solutions to apply. Selecting the option displays a message box containing a list of possible causes for the failure to connect. The text describes ways to fix the problem.

— Note —

This list provides only suggestions. Some suggestions might not be applicable to your debug target.

Chapter 4

Loading Images and binaries

This chapter describes how to load images and binaries during a RealView® Debugger debugging session. It contains the following sections:

- *About loading images and binaries* on page 4-2
- *Loading an executable image* on page 4-4
- *Viewing image details* on page 4-9
- *Loading a binary* on page 4-12
- *Loading multiple images to the same target* on page 4-14
- *Loading symbols only for an image* on page 4-16
- *Replacing the currently loaded image* on page 4-17
- *Loading an executable image on startup* on page 4-18
- *Unloading an image* on page 4-20
- *Deleting the process details for an unloaded image* on page 4-21
- *Reloading an image* on page 4-22
- *Reloading a binary* on page 4-23
- *Changing the format of the disassembly view* on page 4-24
- *Interleaving source in the disassembly view* on page 4-25
- *Opening source files for a loaded image* on page 4-26
- *Saving and closing source files* on page 4-28
- *Hiding line numbers for opened source files* on page 4-29
- *Adding source file search paths for a loaded image* on page 4-31
- *Autoconfiguring search rules for locating source files* on page 4-34.

4.1 About loading images and binaries

When you start RealView Debugger you can begin to use some features such as editing and searching source code. However, to debug images and binaries you must first connect to a suitably configured debug target.

After starting RealView Debugger, and connecting to a debug target, you can:

- Load one or more executable images to begin your debugging session. When you load an image it is added to the Recent Images list.
- Load binary files of various types to specific locations in memory. When you load a binary it is added to the Recent Binaries list. In addition, the type of binary, and the location where it was previously loaded, is shown for each entry.

The lists contain only those images and binaries that you have previously loaded for the type of target shown in the Code window.

The examples in this chapter assume that you are using a Typical installation and that the software has been installed in the default location. If you have changed these defaults, or set the environment variable `RVDEBUG_INSTALL`, your installation differs from that described.

See also:

- *Source file search path*
- *Module naming conventions* on page 4-3
- *Chapter 3 Target Connection*.

4.1.1 Source file search path

When loading an image, RealView Debugger searches for application source file paths using the following sequence:

1. From information contained in the image.
2. From a list of directories specified in any previous settings file that is stored with the image.
3. The current working directory for the source file or files.

————— Note —————

If the current working directory is called Debug or Release, then RealView Debugger looks in the parent directory.

4. If RealView Debugger is unable to locate a source file, then it displays the Source File Location dialog box, where you can specify the path for that source file. The source path is saved in a settings file that is stored in the same location as the image. The file has the name *image_name.ext.apr*. To view the source path, select **Set Source Search Path...** from the **Debug** menu.

See also

- *The current working directory* on page 2-10
- *Adding source file search paths for a loaded image* on page 4-31.

4.1.2 Module naming conventions

When you load an executable image, RealView Debugger creates module names based on the source file names:

- For sources with the .*c, .*c++, .*cp, .*cpp, .cxx, or .ixx extensions, the module name is the source filename without the extension. If the extension is not one of these extensions, then the extension is preserved, and the dot is replaced with an underscore.
- All module names are converted to uppercase by RealView Debugger. Therefore, you must specify module names in uppercase. For example, sample_arm.c is converted to SAMPLE_ARM, and sample_arm.s is converted to SAMPLE_ARM_S.
- If two modules have the same name then RealView Debugger appends an underscore followed by a number to the second module, for example SAMPLE_ARM_1. Additional modules are numbered sequentially, for example, if there is a third module this becomes SAMPLE_ARM_2.

Following this convention avoids any confusion with the C dot operator that indicates a structure reference.

4.2 Loading an executable image

Before you can start debugging your image, you must load it onto the target.

See also:

- *Before you load an image*
- *Loading an executable image with options* on page 4-5
- *Loading an image from the Home Page* on page 4-6
- *Loading an image to a target that supports TrustZone technology* on page 4-6
- *Loading an image to a SoC Designer target* on page 4-7
- *Loading from the Process Control view* on page 4-7
- *Loading an image by dragging and dropping* on page 4-7
- *Considerations when loading an image* on page 4-8.

4.2.1 Before you load an image

Before you load an image:

- Make sure of the following:
 - The target for the current connection shown in the Code window title bar must match the processor type of the image you are trying to load. If they do not match the load fails.
 - The processor or architecture of the target must match the processor or architecture that you specified when you built the image. Although the image might load, it might not run correctly. For example, you cannot run an image built for an ARM966E-S™ processor on an ARM7TDMI® target.
- If you are using *RealView ARMulator® ISS* (RVISS) to simulate an ARM architecture-based debug target, loading an image does not automatically send a reset. To reset a simulated processor, enter a RESET command before you load, or reload, the image.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the RESET command.

4.2.2 Loading an executable image with options

To load an executable image:

1. See *Before you load an image* on page 4-4.
2. Select **Load Image...** from the **Target** menu to display the Load Image dialog box.
Figure 4-1 shows an example:

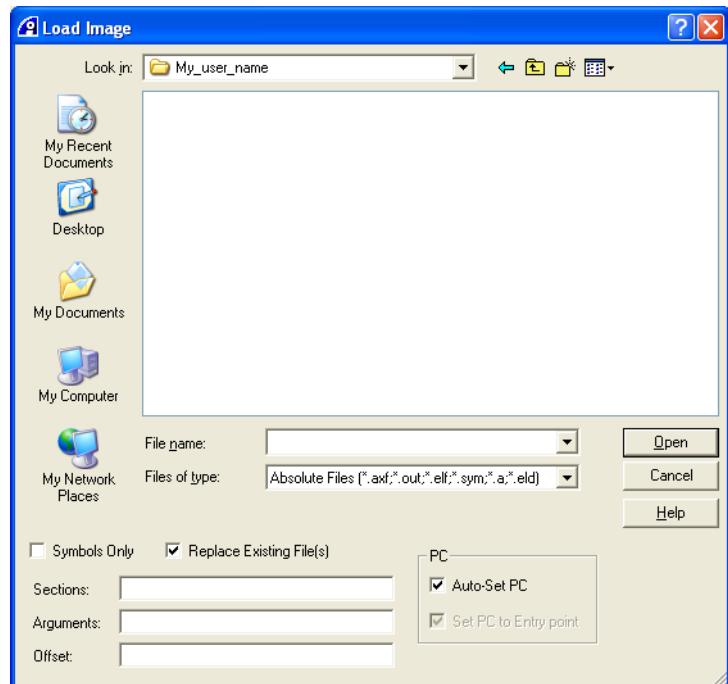


Figure 4-1 Load Image dialog box

3. Locate the executable image that you want to load.
4. If you want to load sections of the image, then enter the sections in the Sections field. This must be a comma-separated list without any spaces, for example, ER_RW,ER_RO,ER_ZI.
5. If your image accepts arguments, then enter the values for the arguments in the Arguments field. This must be a space separated list, for example, 100 2 "Y".
6. If you are loading a relocatable image, enter the address offset in the Offset field.

————— **Note** —————

To load an image to the Normal World of a target that supports TrustZone® technology, you must use the N: address prefix.

7. Click **Open** to load the image.

See also

- *Loading an image to a target that supports TrustZone technology* on page 4-6.

4.2.3 Loading an image from the Home Page

The **Home Page** lists the last 10 images you have loaded. To load an image directly from the **Home Page**:

1. See *Before you load an image* on page 4-4.
2. Select **Home Page** from the **View** menu to display the **Home Page** in the Code window. Figure 4-2 shows an example. The Recent Images are listed at the top of the page.

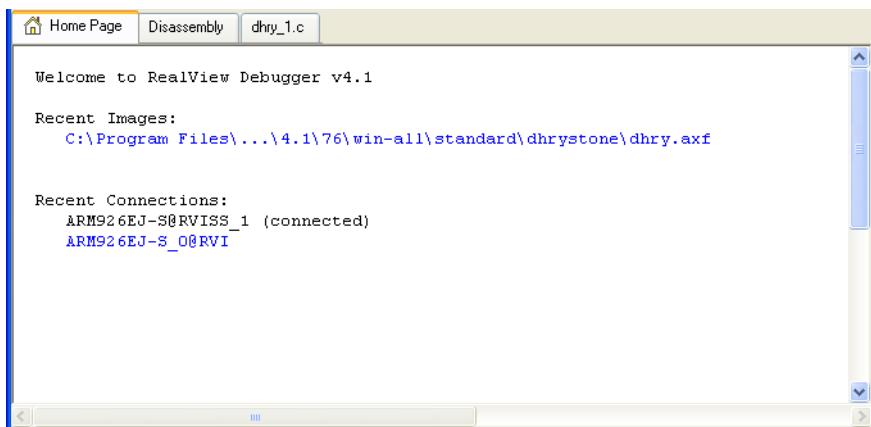


Figure 4-2 Home Page

3. If any images are listed, click the required image to load it to the target shown in the Code window.
4. If no images are listed, click the blue **Load Image...** hyperlink. The Load Image dialog box is displayed.

See also

- *Loading an executable image with options* on page 4-5.

4.2.4 Loading an image to a target that supports TrustZone technology

When you load an image to a target that supports TrustZone technology, then the image is loaded to the current world by default.

To load an image to the other world on a target that supports TrustZone technology:

1. See *Before you load an image* on page 4-4.
2. Select **Load Image...** from the **Target** menu to display the Load Image dialog box. Figure 4-1 on page 4-5 shows an example.
3. Locate the executable image that you want to load.
4. If you want to load sections of the image, then enter the sections in the Sections field. This must be a comma-separated list without any spaces, for example, ER_RW,ER_RO,ER_ZI.
5. If your image accepts arguments, then enter the values for the arguments in the Arguments field. This must be a space separated list, for example, 100 2 "Y".
6. If you are loading a relocatable image, enter the address offset in the Offset field that has the prefix for the other world. For example, if the current world is Secure, then:
 - to load an image at the image entry point in the Normal World, enter **N:0**

- to load a position-independent image at an offset of `0x1000` in the Normal World, enter `N:0x1000`.

Note

To explicitly load an image to an address in the current world, specify the address prefix for the current world, for example, `S:0`.

7. Click **Open** to load the image.

4.2.5 Loading an image to a SoC Designer target

You can load an image to a SoC Designer target using the following methods:

- You can associate an image with each target in a SoC Designer model. Therefore, when you load a model in Carbon SoC Designer Simulator, the images can automatically be loaded to the corresponding targets. To debug these images in RealView Debugger, load the symbols only in RealView Debugger.
- You can load an image to each target in your SoC Designer model directly from RealView Debugger in the usual way.

See also

- *Connecting to a SoC Designer target* on page 3-30
- *Loading an executable image with options* on page 4-5
- *Loading symbols only for an image* on page 4-16
- Carbon SoC Designer Plus, <http://carbondesignsystems.com/Products/SoCDesigner.aspx>.

4.2.6 Loading from the Process Control view

If you have started RealView Debugger and are connected to a debug target, you can load an image for execution from the Process Control view:

1. See *Before you load an image* on page 4-4.
2. Select **Process Control** from the **View** menu to display the Process Control view.
With no image loaded, the view only shows details about the debug target processor and the current location of the PC.
3. If no image is loaded, right-click in the **Process** tab to display the context menu.
If an image is already loaded, you must right-click on the top-level processor entry (for example ARM7TDMI) to display the context menu.
4. Select **Load Image...** from the context menu to display the Select Local File to Load dialog box. Figure 4-1 on page 4-5 shows an example.
5. Complete the entries in the dialog box, to load the required image.

4.2.7 Loading an image by dragging and dropping

With a connection established, you can load an image by dragging the appropriate executable file and dropping it into the **Home Page** tab, **Disassembly** tab, or a source file tab. If successful, this is the same as loading the image using the Select Local File to Load dialog box with the default settings (shown in Figure 4-1 on page 4-5), that is the load auto-sets the PC and overwrites any existing image on the debug target.

4.2.8 Considerations when loading an image

Be aware of the following when loading images:

- An entry for that image is added to the Recent Images list. The lists contain only those images that you have previously loaded for the type of target shown in the Code window.
- RealView Debugger locks the image file while it is loaded. Therefore, you cannot use another application to modify an image file when it is loaded (for example, you cannot rebuild the image). To modify the image file, unload the image.
- If you reload an image by selecting it from the Recent Images list, then any breakpoints, tracepoints, and user-defined macros are cleared.

Automatic loading of source files

If the **Home Page** is selected when you load an image, the source file containing the image entry point for the image is automatically opened.

If the **Disassembly** tab is selected when you load an image, no source file is opened. To open the source file that contains the image entry point, do one of the following:

- Loading a source file using the **Home Page** tab:
 1. Click the **Home Page** tab.
 2. Click the **Locate PC** button on the Debug toolbar. The source file containing the image entry point for the image is opened.
- Loading a source file using the **Disassembly** tab:
 1. Right-click in the **Disassembly** tab to display the context menu.
 2. Select **Locate PC in Source** from the context menu to open the source file containing the image entry point for the image.

Runtime visualization

As you load an image to your debug target, the Code window Status line shows the progress of the load and gives an indication of the percentage complete.

The Status line also shows the Processor status of the debug target:

- Where an image is loaded but not executing, the status shows **Stopped**.
- Where an image is running, the status shows **Running**, together with a moving progress indicator.
- Where the current state of the target is unknown, the status shows **Unknown**. For example it might have been running when the connection was established or it might be disconnected.

If memory mapping is enabled, then the memory map is updated to show details of the image.

See also

- *Unloading an image* on page 4-20
- *How loading an image affects the memory map* on page 9-11
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Considerations when using the Cortex-M1 model* on page B-17.

4.3 Viewing image details

The following sections describe how to manage your application files in the Code window:

- *Viewing image details in the Code window*
- *Viewing image details in the Process Control view on page 4-10.*

The examples assume that you are using a Typical installation and the software has been installed in the default location. If you have changed these defaults, or set the environment variable RVDEBUG_INSTALL, your installation differs from that described here.

4.3.1 Viewing image details in the Code window

If an image is successfully loaded to the target processor, the Code window is updated. Figure 4-3 shows an example Code window with the dhystone example image loaded.

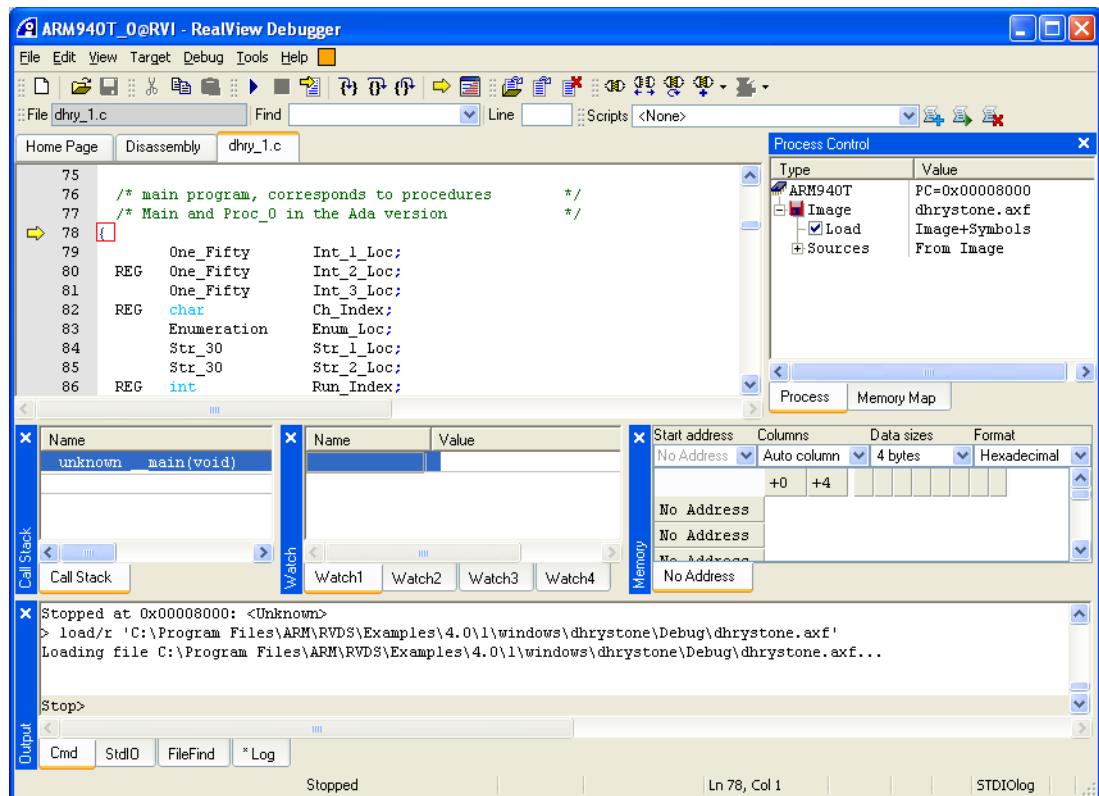


Figure 4-3 Code window with image loaded

Note

Source text coloring and line numbering are enabled by default.

RealView Debugger updates the views with information about the new image, where known. Because you have not started debugging, other views are empty.

When you load an image that contains debugging information, RealView Debugger searches for the corresponding source file associated with the current execution context:

- If there is no debugging information for the current PC, and the PC is at the image entry point, then the source containing `main()` is displayed. This is the situation in this example.

- If the debugging information is at the image entry point, then the source file containing the image entry point is displayed. For example, an assembly source file that contains the ENTRY directive is displayed.

RealView Debugger displays the file as a tab in the Code window.

-  Click the **Locate PC** button on the Debug toolbar to display the line of source code or the line of disassembly related to the address in the PC.

The image was loaded with the **Set PC to Entry point** option set, so execution control is located at the default entry point. This is indicated by a yellow arrow and a red box at line 78 in Figure 4-3 on page 4-9.

Click the **Disassembly** tab to show the disassembly-level view.

4.3.2 Viewing image details in the Process Control view

By default, the side view to the right of the source and disassembly view is the Process Control view. However, if you no longer have the Process Control view visible, select **Process Control** from the **View** menu to display it as a floating view, shown in Figure 4-4.

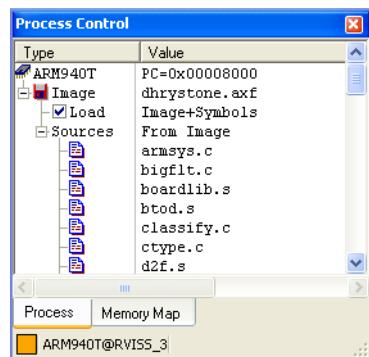


Figure 4-4 Image details in the Process Control view

The Process Control view contains tabs:

Process Displays details of the target processor or, in multiprocessor debugging mode, the current process.

In the example in Figure 4-4, you can see the following entries:

Current process

Shows the target processor and the current state of any running process.

Where the process is stopped, as here, this shows the value of the PC. Where the process is executing, this changes to Run.

Image Details the loaded images:

Load For each image, a check box indicates the load state and what has been loaded, that is image, symbols, or both.

Sources This is a list of the source files extracted from the loaded image.

Memory Map

Displays the memory mapping for the target processor, or the current process.

You can also temporarily change the map settings if required. If you do this:

- your changes to not affect the memory map settings in the configuration files

- when you disconnect from the target, the changes are lost.

The tabs displayed in the Process Control view depend on the debugging mode that you are licensed to use and your current debugging environment. For example, when debugging multithreaded applications, a **Thread** tab is also displayed.

See also

- *Opening source files for a loaded image* on page 4-26
- *Loading from the Process Control view* on page 4-7
- *Refreshing the symbols* on page 4-16
- *Unloading an image* on page 4-20
- *Reloading an image* on page 4-22
- *Chapter 9 Mapping Target Memory*
- *RealView Debugger RTOS Guide*.

4.4 Loading a binary

Binary files usually require that you specify a location when being loaded, for example a Flash binary. Therefore, you cannot load them in the same way as an executable image.

Note

If you want to load a binary to a location in Flash, a Flash memory entry must exist in the memory map.

To load a binary:

1. Start RealView Debugger.
 2. Connect to a target.
 3. To load a binary to a location in Flash, then enable memory mapping.
 4. Select **Load Binary...** from the **Target** menu to display the Load Binary dialog box.
- Figure 4-5 shows an example. The default location is the location of the last image you loaded. In this case, the location of the debug image for the Dhrystone example project.

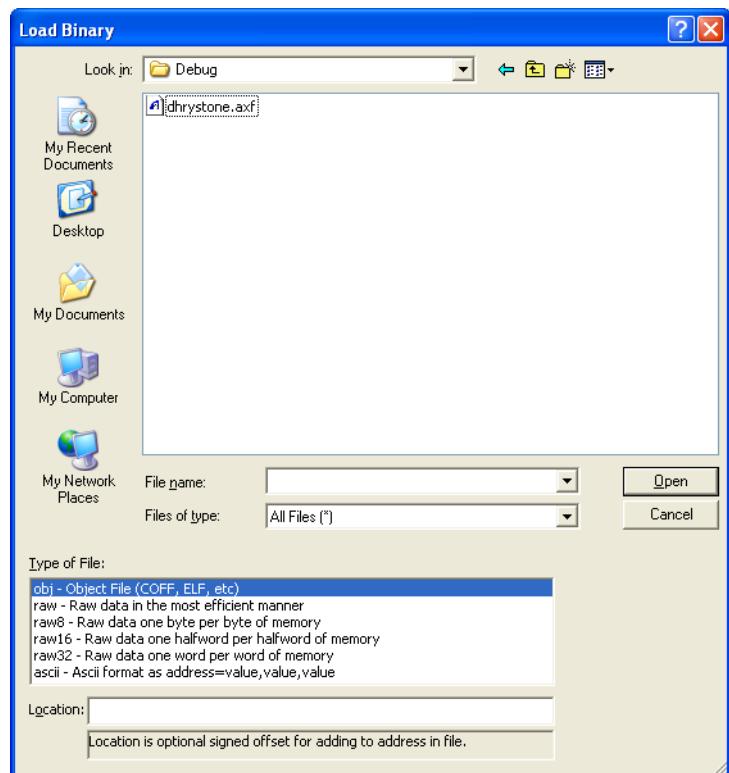


Figure 4-5 Load Binary dialog box

5. In the Look in: field, locate the directory containing the binary file that you want to load.
6. Select the type of binary file that you are loading from the **Type of File** list.
7. Enter the memory location where you want to load the binary in the **Location** field:
 - For files of type **obj** and **ascii**, you can optionally enter as value. The value is an offset that is added to the start address in the file.
 - For other file types, you must enter a value. The value is an absolute memory address.

8. Click **Open** to load the binary to the specified location.

See also:

- *Considerations when loading a binary*
- *Connecting to a target* on page 3-27
- *Enabling memory mapping* on page 9-6
- *Memory map entry* on page 9-9.

4.4.1 Considerations when loading a binary

Be aware of the following:

- When you load a binary no details are shown in the **Process** tab of the Process Control view (see Figure 4-4 on page 4-10). However, the memory map might be updated to show the memory details for the binary.
- If you load a binary to a location in Flash, then the Flash Memory Control dialog box is displayed.
- When you load a binary, an entry for that binary is added to the Recent Binaries list. In addition, the type of binary, and the location where it was previously loaded, is included for each entry. The list contains only those binaries that you have previously loaded for the type of target shown in the Code window.

See also

- Chapter 6 *Writing Binaries to Flash*
- Chapter 9 *Mapping Target Memory*.

4.5 Loading multiple images to the same target

RealView Debugger provides the option to load multiple images to the same debug target, that is where there is only a single connection. This enables you to load, for example, both an executable image and an OS-aware image at the same time.

The procedure described in the next section assumes that you have two images, hello.axf and demo.axf, that do not overlap in memory. For example:

- hello.axf might be built with:
`armcc -g -O2 hello.c -o hello.axf`
- demo.axf might be built with:
`armcc -c -g -O2 demo.c -o demo.o
armlink --ro-base 0x11000 demo.o -o demo.axf`

To load two images to the same debug target:

1. Load the first image, for example hello.axf, with the default load options.
2. Load a second image to the same target, for example demo.axf.

This image must not overlap any part of the first image in memory. You must load the second image as follows:

- a. Select **Load Image...** from the **Target** menu to display the Load File to Target dialog box.
- b. Deselect the **Replace Existing File(s)** check box, to prevent this image replacing the first image.

The **Set PC to Entry point** check box is also deselected. This ensures that the PC still points to the entry point in the first image.

Note

The **Set PC to Entry point** check box must be selected only for the image that contains the entry point, which might not be the first image you load.

- c. Click **Open**.
3. Select **Process Control** from the **View** menu to display the Process Control view.
4. Expand the display to see the process details. Figure 4-6 shows an example:

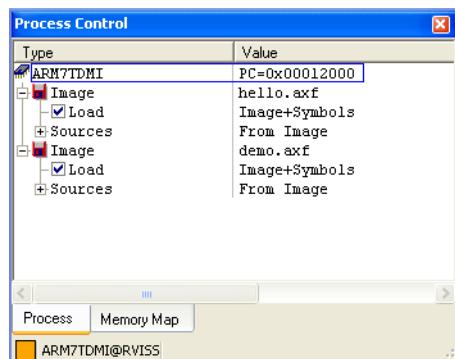


Figure 4-6 Multiple images in the Process Control view

Because no image is currently executing, the Process entry shows the current location of the PC, which was auto-set when the first image was loaded. You can set the PC manually to start debugging or reload the image that you want to test.

Note

RealView Debugger can only keep track of the entry point for a single image. Therefore, if you are working with multiple images, you must set a manual breakpoint at the entry point of any other images you have loaded. This ensures that RealView Debugger is able to debug these images in the usual way.

See also:

- *Loading an executable image* on page 4-4
- *Chapter 8 Executing Images*
- *Chapter 11 Setting Breakpoints*
- *Chapter 10 Changing the Execution Context*
- *Chapter 14 Altering the Target Execution Environment*
- *ARM® Compiler toolchain Using the Compiler*
- *ARM® Compiler toolchain Compiler Reference*
- *ARM® Compiler toolchain Using the Linker*
- *ARM® Compiler toolchain Linker Reference*.

4.6 Loading symbols only for an image

An executable image contains symbolic references, such as function and variable names, in addition to the program code and data. You can choose to load only the symbols for the image, for example, if you are using RealMonitor.

See also:

- *Loading the symbols*
- *Refreshing the symbols*
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43 for more details on using RealMonitor.

4.6.1 Loading the symbols

To load only the symbols for an image:

1. Select **Load Image...** from the **Target** menu to display the Select Local File to Load dialog box. An example is shown Figure 4-1 on page 4-5.
2. Locate the image containing the symbols to be loaded.
3. Select the **Symbols Only** check box. The **Auto-Set PC** and **Set PC to Entry point** check boxes are deselected.
4. Click **Open**. The symbolic references are loaded into the debugger without loading any code or data to the target. You might want to do this if the code and data are already present on the debug target, for example in a ROM device or where you are working with an OS-aware target.

4.6.2 Refreshing the symbols

You can choose to refresh the symbol data for a loaded image during your debugging session. There are two ways to do this for the current process:

- Select **Refresh Symbols** from the **Target** menu.
- In the Process Control view:
 1. Right-click on the **Image** entry for the required image to display the **Image** context menu.
 2. Select **Refresh Symbols** from the available options.

This method is useful if you have multiple images loaded on the same target.

Note

When an image is loaded with symbols, the symbol table is recreated. This automatically deletes any user-defined macros that you have loaded, because these are stored in the symbol table.

4.7 Replacing the currently loaded image

You do not have to unload an image from a debug target before loading a new image for execution.

To load over an existing image:

1. Select **Load Image...** from the **Target** menu to display the Select Local File to Load dialog box. Figure 4-1 on page 4-5 shows an example.
2. Locate the new image to be loaded.
3. Make sure the **Replace Existing File(s)** check box is selected on the Select Local File to Load dialog box, shown in Figure 4-1 on page 4-5. This is the default setting.
4. If you want to have control over whether the PC is set to the image entry point or not, then deselect the **Auto-Set PC** check box.
The **Set PC to Entry point** check box is enabled.
5. Click **Open**.

All details about the first image are removed from RealView Debugger.

4.8 Loading an executable image on startup

You can start RealView Debugger from the command line and specify an image to load automatically. You must also specify a target connection to use, and you can pass arguments to the image and specify any image sections to be loaded.

— Note —

You cannot load a binary on startup. See *Loading a binary* on page 4-12 for details of how to load a binary.

To load an image from the command line, use the following command syntax:

```
rvdebug.exe --init=@target@DebugConfiguration --exec image.axf[;sections][;arg1 arg2 ...]
```

where:

target Specifies the target connection name.

DebugConfiguration

Specifies the name of the Debug Configuration that defines the connection to the target.

image.axf Specifies the image to be loaded. Also include the path name if required.

sections Specifies an optional comma-separated list of sections to load for the image, for example, ER_RO,ER_RW.

arg1 arg2 ...

Specifies an optional space-separated list of arguments to the image.

— Note —

You cannot use this method of specifying arguments to an image for ISSMs.

— Note —

Spaces must not be included between the argument and the qualifier. Where an arguments list is given, quotes must be used.

See also:

- *Examples of loading an image on startup* on page 4-19
- *Connection prompt on load failure* on page 4-19
- Chapter 2 *The RealView Debugger Environment*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the LOAD command.

4.8.1 Examples of loading an image on startup

The following examples show how to use the --init and --exec arguments:

- To establish a connection to an ARM926EJ-S RVISS model, and load an image with sections ER_RO and ER_RW and the argument 5000:

```
rvdebug.exe --init=@ARM926EJ-S@RVISS_1 --exec  
"C:\rvd\images\my_image.axf;ER_RO,ER_RW;5000"
```

If you want to supply arguments, but no sections, leave the sections blank, for example:
"C:\rvd\images\my_image.axf;;5000"

- To establish a connection to a Cortex-A8 model, and load an image without specific sections and without arguments:

```
rvdebug.exe --init=@ARM_Cortex-A8@ISSM --exec C:\rvd\images\my_image.axf
```

- If a pathname includes spaces, it must be enclosed in quotes, for example:

```
rvdebug.exe --init=@ARM_Cortex-A8@ISSM --exec "C:\rvd\my images\my_image.axf"
```

These examples start RealView Debugger, connect to a Cortex-A8 model, and issue a `load/pd/r` command to load the named image to your debug target:

- the `/pd` switch specifies that any error messages are to appear in a dialog box
- the `/r` switch specifies that any image currently loaded on the chosen target is to be replaced by the specified image.

4.8.2 Connection prompt on load failure

If you do not specify a target connection when loading an image from the command line, or the specified connection fails, then RealView Debugger displays the Image Load Failure dialog box. Figure 4-7 shows an example:

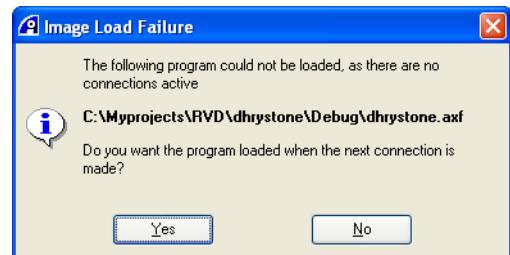


Figure 4-7 Image Load Failure dialog box

To complete the connection, click:

Yes To cause the debugger to wait until you successfully connect to your debug target. When you have connected to the debug target, RealView Debugger then loads the image to that target.

No To start the debugger and cancel the image loading operation.

4.9 Unloading an image

You might want to unload an image for various reasons. For example, if you have rebuilt the image after it was loaded, and you want to clear any debugging features that might not be relevant to the new image, such as breakpoints.

To unload an image:

1. Select **Process Control** from the **View** menu to display the Process Control view, if it is not already visible.
2. Deselect the **Load** check box.

The debug information is removed from RealView Debugger.

Alternatively, select **Unload Image** from the **Target** menu. If more than one image is loaded on the target, an Unload Image dialog box is displayed. Figure 4-8 shows an example. Select one or more images to unload, then click **OK**.

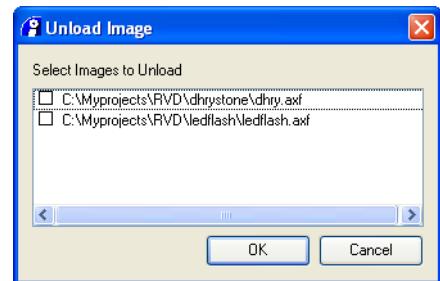


Figure 4-8 Unload Image dialog box

See also:

- *Considerations when unloading an image.*

4.9.1 Considerations when unloading an image

Be aware of the following:

- Any breakpoints and tracepoints that you have set are cleared.
- If any source file are currently opened, they remain available for editing, if required.
- Unloading an image does not affect target memory. It unloads the symbol table (and any macros) and removes debug information from RealView Debugger. However, the image name is retained for reloading.

See also

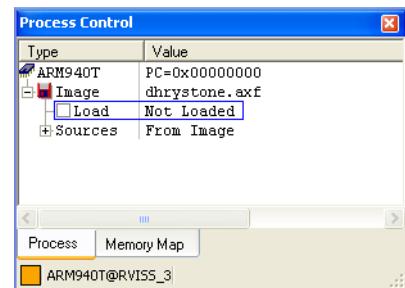
- *Deleting the process details for an unloaded image* on page 4-21.

4.10 Deleting the process details for an unloaded image

When you unload an image, some of the process details, such as the image name, are retained.

To remove all details about an image after you have unloaded it:

1. Select **Process Control** from the **View** menu to display the Process Control view, if it is not already visible.
2. Right-click on the **Image** entry in the Process Control view, as shown in Figure 4-9, to display the **Image** context menu.

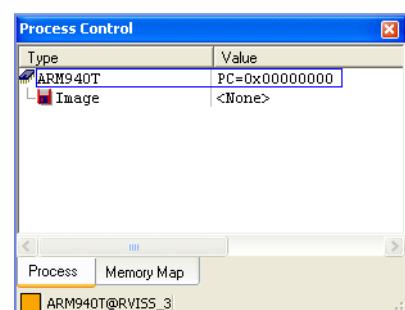


The screenshot shows the 'Process Control' window with a single entry in the tree view: 'Image'. Under 'Image', there are two sub-items: 'Load' (set to 'Not Loaded') and 'Sources' (set to 'From Image'). The 'Value' column next to 'Image' shows 'dhrystone.axf'. At the bottom of the window, there are tabs for 'Process' and 'Memory Map', with 'Process' selected. A status bar at the bottom displays 'ARM940T@RVISS_3'.

Figure 4-9 Unloaded image

3. Select **Delete Entry** from the context menu.

The image details are removed, as shown in Figure 4-10.



The screenshot shows the 'Process Control' window with the same structure as Figure 4-9, but the 'Value' column for 'Image' now shows '<None>'. The other entries ('ARM940T' and its sub-items) remain the same. The tabs and status bar are also identical to Figure 4-9.

Figure 4-10 Deleted image

4.11 Reloading an image

During your debugging session you might have to edit your source code and then recompile. Following these changes, you can reload the image if the old image is still loaded. Reloading refreshes any window displays and updates debugger resources.

To reload an image:

1. Select **Process Control** from the **View** menu to display the Process Control view, if it is not already visible.
2. Right-click on the **Image** entry in the Process Control view to display the **Image** context menu.
3. Select **Reload** from the context menu.

See also:

- *Considerations when reloading images.*

4.11.1 Considerations when reloading images

Be aware of the following:

- If you used the Select Local File to Load dialog box to enter sections and arguments to the image, these are not used when you reload. To reuse the arguments and sections for the new image, do one of the following:
 - Select **Recent Images** from the **Target** menu to reload the image from the Recent Images list. This lists only those images that you have previously loaded for the type of target shown in the Code window.

————— **Note** ————

When you reload an image in this way, then any breakpoints, tracepoints, and user-defined macros are cleared.

- Select **Set PC to Entry Point** from the **Debug** menu. This submits a RESTART command.
- If you reload an image that resides in Flash, then RealView Debugger re-displays the Flash Memory Control dialog box. However, RealView Debugger resets the PC to the image entry point, and you do not have to rewrite the image to Flash. Click **Close** to close the Flash Memory Control dialog box. Alternatively, if you reset the PC to the image entry point, this is avoided.

See also

- *Loading an executable image* on page 4-4
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

4.12 Reloading a binary

During your debugging session you might want to reload a binary.

To reload a binary, select **Recent Binaries** from the **Target** menu to reload the binary from the Recent Binaries list.

The Recent Binaries list contains only those binaries that you have previously loaded for the type of target shown in the Code window. In addition, the type of binary, and the location where it was previously loaded, is shown for each entry.

See also:

- *Considerations when reloading binaries.*

4.12.1 Considerations when reloading binaries

Be aware that if the binary was previously loaded to Flash, then the Flash Memory Control dialog box is displayed.

See also

- Chapter 6 *Writing Binaries to Flash*

4.13 Changing the format of the disassembly view

When working in disassembly view, you can temporarily choose the display format for your code view.

To change the format of the disassembly view:

1. Connect to your target.
2. Load an image, for example dhrystone.axf.
3. Click the **Disassembly** tab.
4. Right-click in the **Disassembly** tab to display the context menu.
5. Select **Disassembly Format...** from the context menu to see the Disassembly Mode selection box, shown in Figure 4-11.

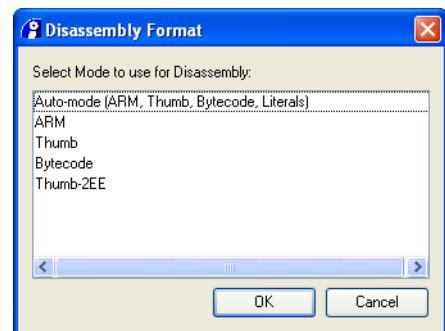


Figure 4-11 Disassembly Mode selection box

Use this selection box to specify how disassembled code is displayed. If you choose Auto-mode, the default format, RealView Debugger displays the code in a format specified by the image contents. If no image information is available, then the current state of the processor is used.

6. Select the required format and click **OK** to confirm your choice. Click **Cancel** to close the selection box without changing the display format.

You can use workspace settings to specify the format used for disassembly.

See also:

- *Disassembler* on page A-3 for details of the Disassembler settings in the DEBUGGER group.

4.14 Interleaving source in the disassembly view

When you view the disassembly, the source is interleaved by default. You can toggle the display of interleaved source as required.

To toggle the interleaving of the source in the disassembly view:

1. Click the **Disassembly** tab in the Code window to display the disassembly view.
2. Right-click in the **Disassembly** tab to display the context menu.
3. Select **Interleave Source** from the context menu.

The source is interleaved with the disassembly view.

4.15 Opening source files for a loaded image

You can open a source file from any location. However, when you load an image onto your target, RealView Debugger lists the sources for the image in the Process Control view. You can also open a source file using this source list.

See also:

- *Opening a source file from any location*
- *Opening a file from the image file list*
- *Locating a source file for an image in the Process Control view* on page 4-27.

4.15.1 Opening a source file from any location

To open a source file:

1. Select **Open...** from the **File** menu to display the Select File to Open dialog box.
2. Locate the directory containing your source files.
3. Select the required source file.
4. Click **Open**.

The Select File to Open dialog box closes. The chosen source file is opened, and the cursor is placed at the start of that file.

5. If an open source file has unsaved edits, you are prompted to save the source file:
 - Click **Yes** to save the changes.
 - Click **No** if you do not want to save the changes.

The source file is opened into a tab in the Code window. If the file is already open:

- RealView Debugger does not re-open the file if it has unsaved edits
- re-opens the file into the existing tab for that file.

See also

- *Saving and closing source files* on page 4-28.

4.15.2 Opening a file from the image file list

To open a source file using the list of sources for a loaded image:

1. Select **Process Control** from the **View** menu to display the Process Control view, if it is not already visible.
2. Expand the Sources entry in the **Process** tab. The list of sources for the image is displayed.
3. Locate the required source file.
4. To open the source file, either:
 - Right-click on the source file, and select **Open File** from the context menu.
 - Double-click on the source file.

The chosen source file is opened, and the cursor is placed at the start of that file.

5. If an open source file has unsaved edits, you are prompted to save the source file:
 - Click **Yes** to save the changes.
 - Click **No** if you do not want to save the changes.

The source file is opened into a tab in the Code window. If the file is already open:

- RealView Debugger does not re-open the file if it has unsaved edits
- re-opens the file into the existing tab for that file.

See also

- *Image details in the Process Control view* on page 4-10
- *Saving and closing source files* on page 4-28
- *Locating a source file for an image in the Process Control view*.

4.15.3 Locating a source file for an image in the Process Control view

You can locate a source file for an image in the Process Control view by scrolling the view contents. However, if your image has been built from a large number of source files, you can quickly locate a source file by using the type ahead facility.

Using the type ahead facility to locate a source file

To use the type ahead facility:

1. Select the image containing the source file that you want to locate.
2. Type the first letter of the source file.
RealView Debugger locates the first matching source file. If the Sources entry is collapsed, then RealView Debugger expands it first.
3. If more than one source file starts with the same letter, type additional letters to locate the appropriate source file. Each letter you type is added to the type ahead buffer.

When using this facility, the type ahead buffer is not case sensitive and is limited to 128 characters. Do one of the following to clear the buffer:

- select a different item either by left-clicking on an item or by using the arrow keys
- press Home to move to the top of the view
- press Esc.

4.16 Saving and closing source files

You can save and close source files individually, or all together in a single operation.

See also:

- *Closing one or more source files*
- *Saving and closing selected files in a single operation.*

4.16.1 Closing one or more source files

To closing one or more source files:

1. Right-click on a source file tab.
2. Select the required option from the context menu:

Close Closes the source file you used to display the context menu.

Close All Closes all open source files.

Close Other

Closes all source files, except for the source file you used to display the context menu.

If you have modified a source file, then you are prompted to save the file before closing it.

See also

- *Opening source files for a loaded image* on page 4-26.

4.16.2 Saving and closing selected files in a single operation

To save and close selected files in a single operation:

1. Select **Save/Close Multiple...** from the **File** menu. The Save Multiple dialog box is displayed, and the source file that is currently visible in the Code window is selected by default.

————— **Note** —————

This option is available only when a source file tab is selected.

2. Select the source files that you want to save or close. Use the **AllOn** and **AllOff** buttons to select or deselect all source files if required.

3. Choose the required operation:

- Click **Close All** to close all the selected source files.
If you have modified a source file, then you are prompted to save the file before closing it.
- Click **Save** to save all the selected source files.
- Click **Convert** to convert files from DOS to UNIX by removing end-of-line markers, or adding end-of-line markers to convert UNIX files to DOS format. The current file type is displayed in the Type column.

See also

- *Opening source files for a loaded image* on page 4-26.

4.17 Hiding line numbers for opened source files

The display of line numbers in your open source files is enabled by default.

See also:

- *Turning off line numbering for the current debugging session only*
- *Turning off line numbers permanently.*

4.17.1 Turning off line numbering for the current debugging session only

To temporarily turn off line numbering for source files that are open:

1. Select **Edit** → **Advanced** → **Show Line Numbers** from the Code window main menu.

The line numbers are no longer displayed in the source code view.

See also

- *Opening source files for a loaded image* on page 4-26.

4.17.2 Turning off line numbers permanently

To permanently view the line numbers in source files that are open:

1. Select **Options...** from the **Tools** menu to display the Options window. Figure 4-12 shows an example. This window contains your global workspace settings.

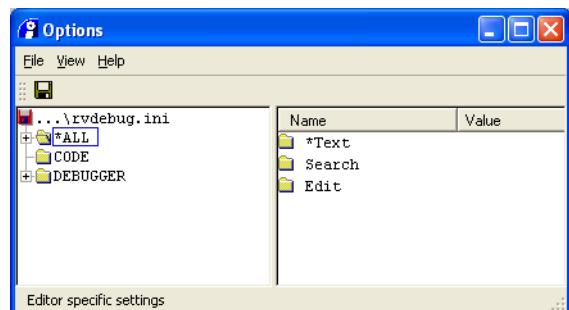


Figure 4-12 Settings Window

2. Expand the ALL entry.
3. Select the Edit entry. The Edit settings are displayed in the right pane, shown in Figure 4-13.

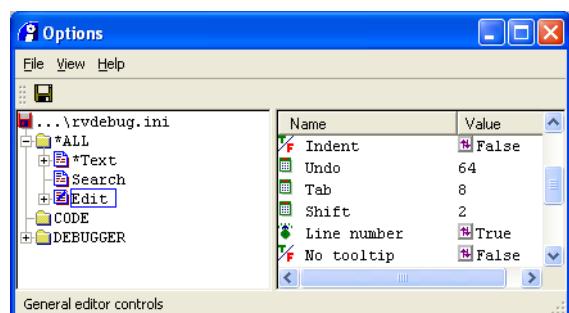


Figure 4-13 The line number workspace setting

4. Change the Line number setting to **False**.

5. Select **Save and Close** from the **File** menu to save the changes and close the Settings Window.

A prompt is displayed to warn you that some settings only take effect when you restart RealView Debugger. You must restart RealView Debugger after changing the Line number setting for the change to take effect.

6. Click **OK**.

Line numbers are not displayed for all subsequent debugging sessions.

See also

- *Opening source files for a loaded image* on page 4-26
- *Chapter 17 Configuring Workspace Settings*.

4.18 Adding source file search paths for a loaded image

If RealView Debugger is unable to locate a source file for your image, you can add the location of the sources to the source file search path.

See also:

- *Adding a source search path*
- *Adding a source mapping* on page 4-32.

4.18.1 Adding a source search path

To add source file search paths for a loaded image:

1. Select **Set Source Search Path...** from the **Debug** menu to display the Source Search and Mappings dialog box. Figure 4-14 shows an example. The **Source Search Path** tab is selected by default.

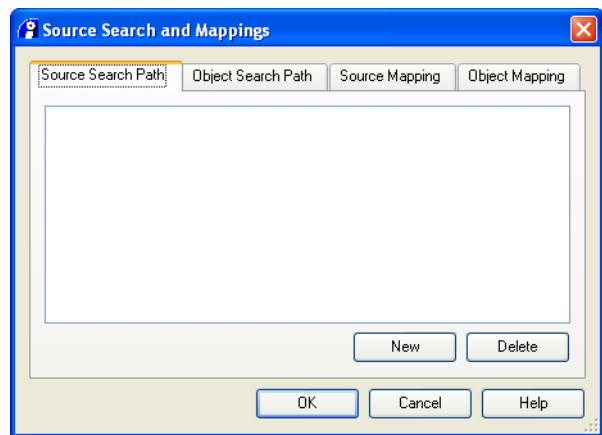


Figure 4-14 Source Search Paths dialog box

2. Click **New** to display the **Browse For Folder** dialog box.
3. Locate the directory containing the source files.
4. Click **OK**.

The source search path is added to the list of paths, as shown in Figure 4-15.

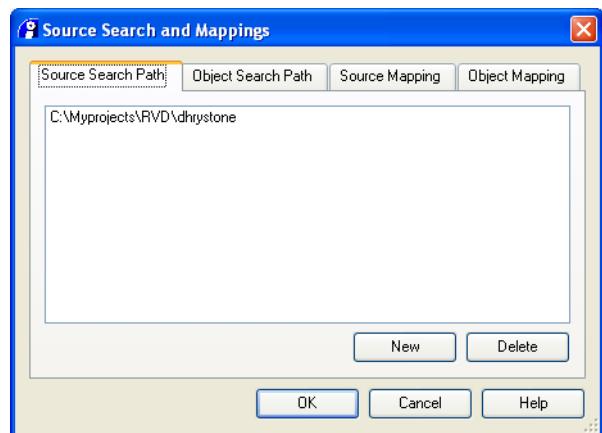


Figure 4-15 Source search path list

5. If your sources are distributed between a number of directories, then repeat the previous steps for each directory.

6. When you have located all the search paths for the image, click **OK** to close the Source Search and Mappings dialog box.

— Note —

The **Object Search Path** and **Object Mapping** tabs are used when debugging OS-aware applications, and provides mappings to objects, such as shared libraries.

The search paths that you define are saved in a settings file that is stored in the same location as the image. This file has the same name as your image, and has the .apr file extension. For example, if you are using the dhystone.axf image, the settings file is called dhystone.axf.apr. The settings in this file are re-applied when you next load the image.

See also

- *RealView Debugger RTOS Guide.*

4.18.2 Adding a source mapping

If an image contains source path details, but that source path does not exist on your workstation, you can specify a source mapping. This maps the source path information in your image to the source path on your workstation.

To add a source mapping for a loaded image:

1. Select **Set Source Search Path...** from the **Debug** menu to display the Source Search and Mappings dialog box. Figure 4-14 on page 4-31 shows an example.
2. Click the **Source Mapping** tab.
3. Click **New** to display the New Mapping dialog box. Figure 4-16 shows an example:

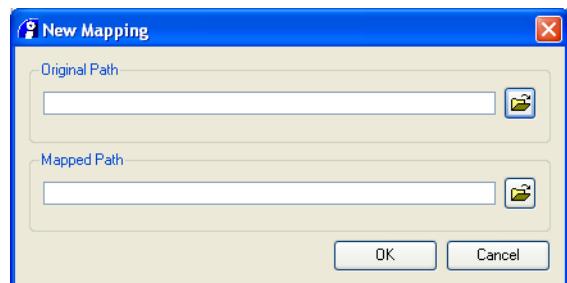
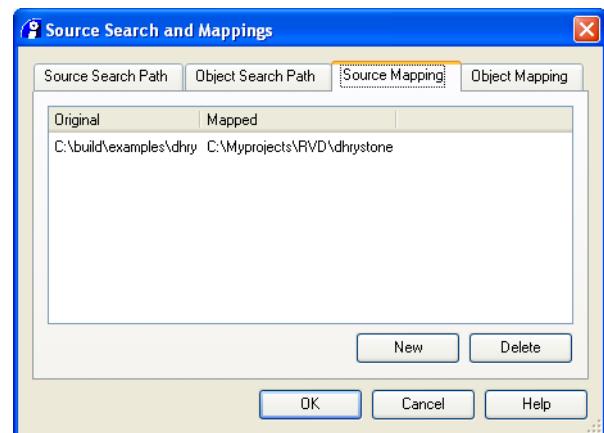


Figure 4-16 New Mapping dialog box

4. Specify the Original Path (that is, the path to the original source tree for the image). To do this, either:
 - enter the path name in the field
 - click the open file button to locate the required file.
5. Specify the Mapped Path (that is, the path to the new source tree for the image). To do this, either:
 - enter the path name in the field
 - click the open file button to locate the required file.
6. Click **OK**.

The source mapping is added to the list of mappings, as shown in Figure 4-17 on page 4-33.

**Figure 4-17 Source Mapping tab**

7. If your sources are distributed between a number of directories, then repeat the previous steps for each directory.
8. When you have specified all the source mappings for the image, click **OK** to close the Source Search and Mappings dialog box.

Note

The **Object Search Path** and **Object Mapping** tabs are used when debugging OS-aware applications, and provides mappings to objects, such as shared libraries.

The search mappings that you define are saved in a settings file that is stored in the same location as the image. This file has the same name as your image, and has the .apr file extension. For example, if you are using the dhrystone.axf image, the settings file is called dhrystone.axf.apr. The settings in this file are re-applied when you next load the image.

See also

- *RealView Debugger RTOS Guide*.

4.19 Autoconfiguring search rules for locating source files

If RealView Debugger cannot find a source file for the loaded image, then it automatically displays the Source File Location dialog box, where you can specify the new path for that source file. This might happen when you attempt to:

- scope to a location in a source file
- view the source file at a location.

See also:

- *Specifying the source file location*
- *Viewing the source mappings* on page 4-35.

4.19.1 Specifying the source file location

To autoconfigure the source search rules:

1. At the Source File Location dialog box, shown in Figure 4-18, either:
 - Edit the path name shown in the dialog box. The path name shown is that obtained from the image.
 - Click the open file button to locate the required file.

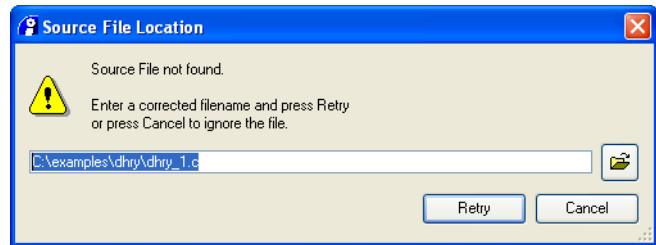


Figure 4-18 Source File Location dialog box

2. Click **Retry**:
 - If RealView Debugger fails to locate the file, the Source File Location dialog box remains open. Correct the path name and try again.
 - If RealView Debugger successfully locates the file, the Source File Location dialog box closes automatically.

The search mapping that you specify is saved in a settings file that is stored in the same location as the image. This file has the same name as your image, and has the .apr file extension. For example, if you are using the dhystone.axf image, the settings file is called dhystone.axf.apr. The settings in this file are re-applied when you next load the image. You can view the source mappings with the Source Search and Mappings dialog box.

See also

- *Viewing the source mappings* on page 4-35.

4.19.2 Viewing the source mappings

To view the source mappings for a loaded image:

1. Select **Set Source Search Path...** from the **Debug** menu to display the Source Search and Mappings dialog box. Figure 4-14 on page 4-31 shows an example.
2. Click the **Source Mapping** tab. Figure 4-19 shows an example:

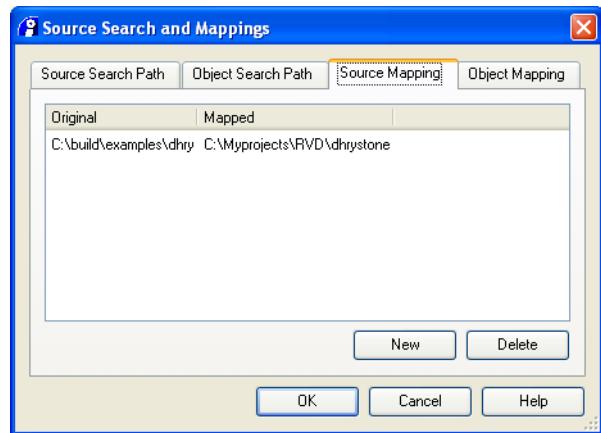


Figure 4-19 Source Mapping tab

— Note —

The **Object Search File** and **Object Mapping** tabs are used when debugging OS-aware applications, and provides mappings to objects, such as shared libraries.

See also

- *Adding a source mapping* on page 4-32
- *RealView Debugger RTOS Guide*.

Chapter 5

Navigating the Source and Disassembly Views

This chapter describes how to navigate the source and disassembly views during a debugging session. It contains the following sections:

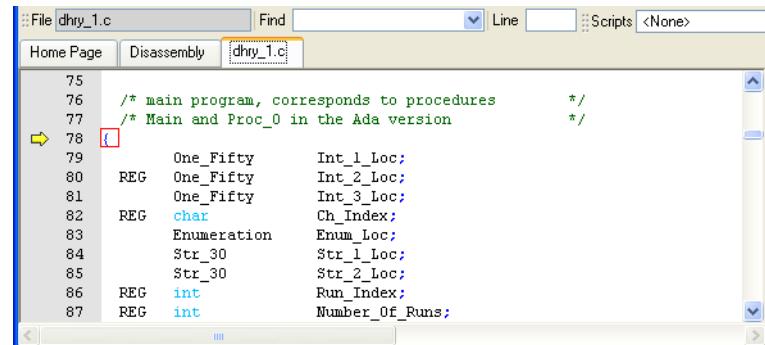
- *About navigating the source and disassembly views* on page 5-2
- *Viewing the selected location in the opposite code view* on page 5-3
- *Locating the lowest address in memory of a module* on page 5-4
- *Locating the line of code using a symbol in the source view* on page 5-6
- *Locating the address of a label in the disassembly view* on page 5-7
- *Locating the destination of a branch instruction* on page 5-8
- *Locating a function* on page 5-9.

5.1 About navigating the source and disassembly views

RealView® Debugger enables you to navigate to various parts of your source code or disassembly views. You can do this by specifying:

- An explicit line number or a symbol in the source view. If the destination is in another source file, then that source file is opened.
- An explicit address or a label in the disassembly view.

The Code window contains a **Disassembly** tab to show the disassembly view for your target. When you load an image containing debug information, the source file containing the image entry point is opened. Figure 5-1 shows an example:



The screenshot shows the RealView Debugger's Code window. The title bar says "File dhry_1.c". Below it is a toolbar with buttons for "Home Page", "Disassembly", and "dhry_1.c". The main area has two panes. The left pane is the "Source View" showing C code. The right pane is the "Disassembly View" showing assembly code. A red box highlights the line number 78 in the source code pane. The assembly code pane shows the following lines:

```

75      /* main program, corresponds to procedures */
76      /* Main and Proc_0 in the Ada version */
77
78      One_Fifty    Int_1_Loc;
79      REG   One_Fifty   Int_2_Loc;
80      One_Fifty   Int_3_Loc;
81      REG   char       Ch_Index;
82      Enumeration  Enum_Loc;
83      Str_30      Str_1_Loc;
84      Str_30      Str_2_Loc;
85      REG   int        Run_Index;
86      REG   int        Number_Of_Runs;
87

```

Figure 5-1 Source code and disassembly views

5.2 Viewing the selected location in the opposite code view

You can view the disassembly corresponding to a selected line of source, or view the line of source corresponding to a disassembled instruction.

See also:

- *Viewing the disassembly corresponding to the selected line of source*
- *Viewing the line of source corresponding to a selected disassembly instruction.*

5.2.1 Viewing the disassembly corresponding to the selected line of source

To view the disassembly corresponding to the selected line of source:

1. Connect to your target.
2. Load an image, for example dhystone.axf. The source file containing the image entry point is opened.
3. In the source code view, locate the required line of source.
For the dhystone.axf example:
 - a. Scroll down to line 149.
 - b. Right-click on Proc_5 to display the context menu.
 - c. Select **Locate Line...** from the context menu.
 - d. Click **Set** in the dialog box. The first line of code in Proc_5() is located (line 378).
4. Right-click on the required line of code to display the context menu (line 378 in this example).
5. Select **Locate Disassembly** from the context menu. The code view changes to the **Disassembly** tab, with the corresponding instruction highlighted.

5.2.2 Viewing the line of source corresponding to a selected disassembly instruction

To view the line of source corresponding to a selected disassembly instruction:

1. Connect to your target.
2. Load an image, for example dhystone.axf. The source file containing the image entry point is opened.
3. In the **Disassembly** tab, locate the required instruction.
For the dhystone.axf example, scroll down to the instruction at address 0x80C0.
4. Right-click in the margin for the required instruction to display the context menu.
5. Select **Locate Source** from the context menu. The code view changes to the source code tab, with the corresponding line of source highlighted.

5.3 Locating the lowest address in memory of a module

You can locate the lowest address that a module occupies in memory.

To locate the lowest address that a module occupies in memory:

1. Connect to your target.
2. Load an image, for example dhystone.axf.
3. Select **Symbols** from the **View** menu to display the Symbols view. Figure 5-2 shows an example:

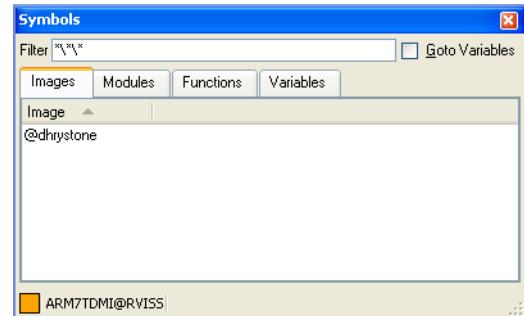


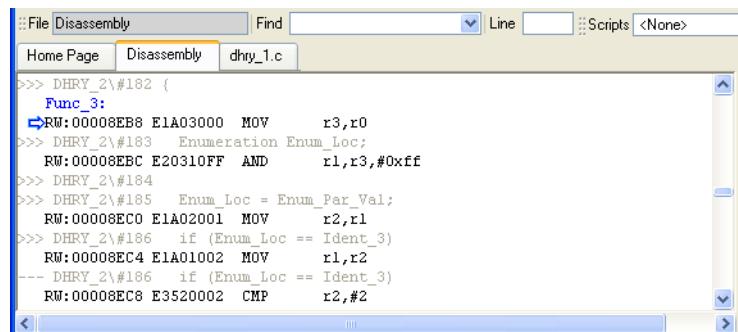
Figure 5-2 Symbols view

4. Click the **Modules** tab to view the modules for the image. Figure 5-3 shows an example:

Symbols		
<input type="text" value="*DHRY*"/> Filter <input type="checkbox"/> Goto Variables		
Module Name	Filename	Image
<Global>	<None>	@dhystone
BIGFLT	../../bigflt.c	@dhystone
BTOD_S	../../btod.s	@dhystone
CTYPE	../../ctype.c	@dhystone
D2F_S	../../d2f.s	@dhystone
DCHECK_S	../../dcheck.s	@dhystone
DCHECK1_S	../../dcheck1.s	@dhystone
DDIV_S	../../ddiv.s	@dhystone

Figure 5-3 Modules in the dhystone.axf image

5. Double-click on the module name you want to locate to display the **Functions** tab.
For example, double-click on the DHRY_2 module.
- Note**
- You can use the Filter field to apply a filter on the complete list of modules. Enter the characters that are common to the required modules. For example, to list all modules that begin with the letters DHRY, then enter ***DHRY***.*.
6. Sort the list of functions by Address.
 7. Double-click on the function name that has the lowest address.
For example, double-click on the Func_3 function. The first location of the module is displayed (at address 0x8EB8 in this example). Figure 5-4 on page 5-5 shows an example:



The screenshot shows a debugger's assembly view. The title bar says "File Disassembly Find Line Scripts <None>". The tab bar has "Home Page" (selected), "Disassembly" (selected), and "dhry_1.c". The assembly code is as follows:

```
>>> DHRY_2\#182 {
    Func_3:
    <RW:00008EB8 E1A03000 MOV      r3,r0
    >>> DHRY_2\#183     Enumeration Enum_Loc;
    <RW:00008EBC E20310FF AND      r1,r3,#0xff
    >>> DHRY_2\#184
    >>> DHRY_2\#185     Enum_Loc = Enum_Par_Val;
    <RW:00008EC0 E1A02001 MOV      r2,r1
    >>> DHRY_2\#186     if (Enum_Loc == Ident_3)
    <RW:00008EC4 E1A01002 MOV      r1,r2
    --- DHRY_2\#186     if (Enum_Loc == Ident_3)
    <RW:00008EC8 E3520002 CMP      r2,#2
```

Figure 5-4 Locating a module

5.4 Locating the line of code using a symbol in the source view

You can locate a line of code for a symbol in the source that you are currently viewing in the Code window. For example, you might want to locate the definition of a called function.

To locate a line of code:

1. Connect to your target.
2. Load an image, for example dhystone.axf.
3. Click the **Locate PC** button on the Debug toolbar to view the source. In this example, the PC is at the entry point at line 78 of file dhry_1.c, marked with yellow arrow and a red box.
4. Locate the line of code containing the symbol you want to locate:
 - a. Scroll down to line 149 in the source file dhry_1.c.
 - b. Right-click on the symbol to display the context menu. For example, right-click on Proc_5.
 - c. Select **Locate Line...** from the context menu to display the Prompt dialog box. The chosen symbol is inserted in the field.
 - d. Click **Set**. The code view from the specified line is displayed. A blue bordered arrow identifies the line. Figure 5-5 shows an example:

```

File: dhry_1.c Find: Line: Scripts: <None>
Home Page Disassembly dhry_1.c
374 Proc_5 () /* without parameters */
375 *****/
376 /* executed once */
377 {
378     Ch_1_Glob = 'A';
379     Bool_Glob = false;
380 } /* Proc_5 */
381
382
383     /* Procedure for the assignment of structures,
384     /* if the C compiler doesn't support this feature */
385 #ifdef NOSTRUCTASSIGN
386 memcpy (d, s, 1)

```

Figure 5-5 Located line of source

5.5 Locating the address of a label in the disassembly view

You can locate the address of a label in the disassembly view.

To locate the address of a label:

1. Connect to your target.
2. Load an image, for example `dhystone.axf`.
3. Click the **Disassembly** tab to view the disassembly. The PC is at the entry point at address `0x00008000`, marked with a red box and yellow arrow.
4. Locate the address of the label:
 - a. Right-click in the **Disassembly** tab to display the context menu.
 - b. Select **Locate Address...** from the context menu.
 - c. Enter the required address. For example, enter the address **0x8480**.
 - d. Click **Set**. The disassembly view from the specified location is displayed. A blue bordered arrow identifies the location.

5.6 Locating the destination of a branch instruction

You can locate the destination of a branch instruction.

To locate the destination of a branch instruction:

1. Connect to your target.
2. Load the required image, for example dhystone.axf. The current scope is at the PC.
3. Click the **Disassembly** tab to view the disassembly. Figure 5-6 shows an example:

```

File Disassembly Find Line Scripts <None>
Home Page Disassembly dhhy_1.c
main:
00008000 EB000000 BL scatterload_rt2 <0x8008>
00008004 EB0007B0 BL __rt_entry <0x9ecc>
__scatterload_rt2:
00008008 E28F002C ADR r0,(pc)+0x34 ; 0x803c
0000800C E8900C00 LDM r0,{r10,r11}
00008010 E08AA000 ADD r10,r10,r0
00008014 E08BB000 ADD r11,r11,r0
00008018 E24A7001 SUB r7,r10,#1
__scatterload_null:
0000801C E15A000B CMP r10,r11
00008020 1A000000 BNE 0x8028 <__scatterload_n>

```

Figure 5-6 Disassembly view showing branch instructions

4. Right-click on the label of the branch instruction, to display the context menu. In the example shown in Figure 5-6, right-click on the `__rt_entry` label.
5. Select **Locate Address...** from the context menu to display the Prompt dialog box together with the chosen label.
6. Click **Set**. The disassembly view changes to show the location of the chosen label, at address `0x9ECC` for this example. A blue bordered arrow identifies the location. Figure 5-7 shows an example:

```

File Disassembly Find Line Scripts <None>
Home Page Disassembly dhhy_1.c
__rt_entry:
RW:00009ECC E1A00000 MOV r0,r0
RW:00009ED0 EB000A79 BL __rt_stackheap_init <0xc8bc>
RW:00009ED4 E92D0003 PUSH {r0,r1}
RW:00009ED8 EB000ADD BL __platform_post_stackheap_init <0xca54>
RW:00009EDC E8BD0003 POP {r0,r1}
RW:00009EE0 EB000A99 BL __rt_lib_init <0xc94c>
RW:00009EE4 E92D000F PUSH {r0-r3}
RW:00009EE8 EB000ADC BL __platform_post_lib_init <0xca60>
RW:00009EEC E8BD000F POP {r0-r3}
RW:00009EF0 EBFFF8C3 BL main <0x8204>
RW:00009EF4 EB000A4C BL exit <0xc82c>

```

Figure 5-7 Navigated to branch destination

5.7 Locating a function

You can locate the start of a function without having to search or scroll through your code to find the function.

See also:

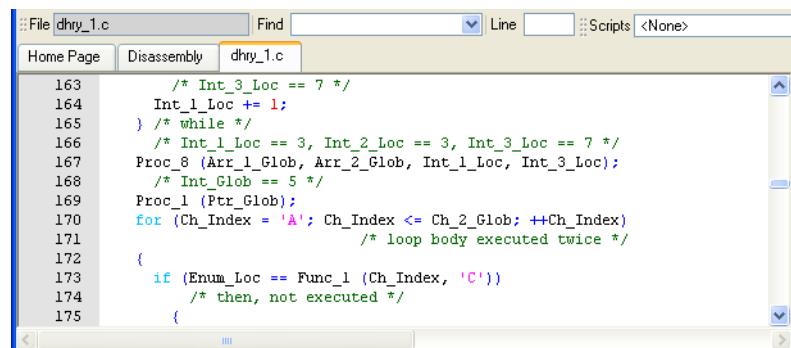
- [Locating a function using the source code view](#)
- [Locating a function using the Symbols view on page 5-10](#)
- [Locating a member function in a C++ class on page 5-12.](#)

5.7.1 Locating a function using the source code view

To locate the start of a function:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
3. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (dhry_1.c in this example).
4. Locate a line in your source where the function is called.

For the dhystone.axf image, Figure 5-8 shows a call to the Proc_8() function at line 167 in dhry_1.c.



```

File dhry_1.c Find Line Scripts <None>
Home Page Disassembly dhry_1.c
163     /* Int_3_Loc == 7 */
164     Int_1_Loc += 1;
165 } /* while */
166 /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
167 Proc_8 (Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
168 /* Int_Glob == 5 */
169 Proc_1 (Ptr_Glob);
170 for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
171 {
172     /* loop body executed twice */
173     if (Enum_Loc == Func_1 (Ch_Index, 'C'))
174     {
175         /* then, not executed */
176     }
177 }
178 
```

Figure 5-8 Call to Proc_8() function

5. Right-click on the Proc_8 function name to display the context menu.
6. Select **Locate Line...** from the context menu to display the Prompt dialog box together with the chosen function name.
7. Click **Set**. The source view changes to show the location of the chosen function, at line 93 in dhry_2.c, as shown in Figure 5-9 on page 5-10. Because the function is defined in dhry_2.c, the file is opened if it is not already open.

```

File dhry_2.c Find Line Scripts <None>
Home Page Disassembly dhry_2.c dhry_1.c
90 Arr_2_Dim     Arr_2_Par_Ref;
91 int           Int_1_Par_Val;
92 int           Int_2_Par_Val;
93 {
94     REG One_Fifty Int_Index;
95     REG One_Fifty Int_Loc;
96
97     Int_Loc = Int_1_Par_Val + 5;
98     Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
99     Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
100    Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
101    for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)

```

Figure 5-9 Proc_8() function located in source view

5.7.2 Locating a function using the Symbols view

To locate a function using the Symbols view:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
3. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (dhry_1.c in this example).
4. Select **Symbols** from the **View** menu to display the Symbols view. Figure 5-10 shows an example:

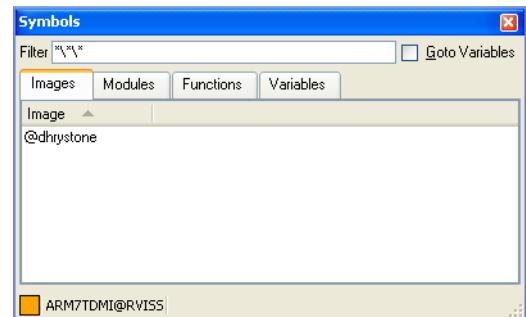


Figure 5-10 Symbols view

5. Click the **Functions** tab to view the functions for the image. Figure 5-11 shows an example:

Function Name	Address	Scope	Module	Image
clock	0x000091AC	Public	SYSAPP	@dhystone
exit	0x0000BE8C	Public	STDLIB	@dhystone
fflush	0x0000C020	Public	STDIO	@dhystone
fgetc	0x0000CB90	Public	STDIO	@dhystone
fopen	0x0000BD74	Public	STDIO	@dhystone
fputc	0x0000CBB8	Public	STDIO	@dhystone
free	0x0000000000000000	Public	HEAD1	@dhystone

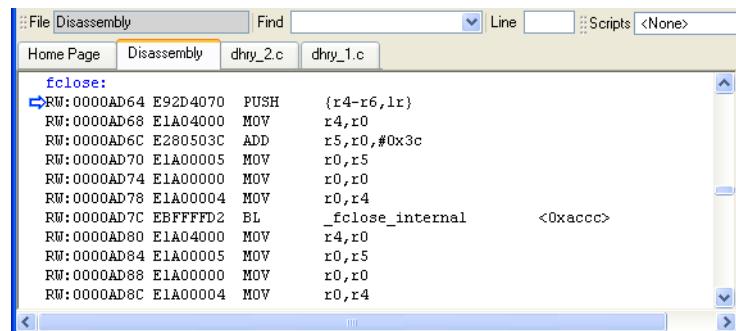
Figure 5-11 Functions in the dhystone.axf image

Note

You can use the Filter field to apply a filter on the complete list of functions. Enter the characters that are common to the required functions.

For example, to list all functions that begin with the letter F, then enter `**\F` or `**\f`.

6. Right-click on the function name to display the context menu. For example, right-click on the `fclose` function.
7. Select **View Disassembly** from the context menu. The disassembly view changes to show the chosen function. Figure 5-12 shows an example:



The screenshot shows a debugger's disassembly window. The title bar includes tabs for "File", "Disassembly", "Find", "Line", "Scripts", and "None". Below the title bar, there are tabs for "Home Page", "Disassembly", "dhy_2.c", and "dhy_1.c", with "Disassembly" currently selected. The main pane displays assembly code for the `fclose` function. The first instruction is highlighted with a blue arrow pointing to it. The assembly listing starts with:

```

fclose:
    <RW:0000AD64 E92D4070> PUSH {r4-r6,lr}
    <RW:0000AD68 E1A04000> MOV r4,r0
    <RW:0000AD6C E280503C> ADD r5,r0,#0x3c
    <RW:0000AD70 E1A00005> MOV r0,r5
    <RW:0000AD74 E1A00000> MOV r0,r0
    <RW:0000AD78 E1A00004> MOV r0,r4
    <RW:0000AD7C EBFFFFD2> BL _fclose_internal <0xaccc>
    <RW:0000AD80 E1A04000> MOV r4,r0
    <RW:0000AD84 E1A00005> MOV r0,r5
    <RW:0000AD88 E1A00000> MOV r0,r0
    <RW:0000AD8C E1A00004> MOV r0,r4

```

Figure 5-12 Navigate to a function

5.7.3 Locating a member function in a C++ class

For a member function of a parent class that is both declared and defined, you can quickly locate that function in your source.

To locate a declared and defined member function:

1. Connect to your target.
2. Load the required image, for example `shapes.axf`.
3. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (`shapes.cpp` in this example).
4. Select **Classes** from the **View** menu to display the Classes view. Figure 5-13 shows an example:

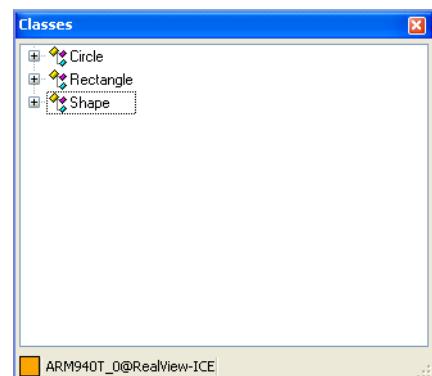


Figure 5-13 Classes view

5. Expand the required class to show the components of that class.
6. Right-click on the required function to display the context menu.
7. Select **Find Definition** from the context menu:
 - If the function has a single declaration, then your code view changes to show the location.
 - If the function has multiple declarations, then a List Selection dialog box is displayed showing the declarations. Deselect all declarations except for the one that is of interest, and click **OK**. The code view changes to show the location.

See also

- *Identifying the components of a class in the Classes view* on page 13-23 for details of a filled stack.

Chapter 6

Writing Binaries to Flash

This chapter describes how to write complete binaries to Flash, or to write to specific locations in Flash. It contains the following sections:

- *About writing binaries to Flash* on page 6-2
- *Writing a binary to Flash* on page 6-4
- *Writing to specific locations in Flash memory* on page 6-6
- *Viewing information about the Flash memory* on page 6-8
- *Operations available when writing to Flash* on page 6-10.

6.1 About writing binaries to Flash

The following sections give an overview of writing binaries into the Flash device on your debug target:

- *Requirements for writing binaries to Flash*
- *Flash Method files*
- *Flash examples* on page 6-3
- *Basic procedure for programming a binary into Flash* on page 6-3.

6.1.1 Requirements for writing binaries to Flash

To use RealView® Debugger to control Flash memory on your chosen debug target, you must:

- Have access to an appropriate *Flash MMethod* (FME) file.
- Configure your connection settings to specify:
 - a memory map of your debug target that includes a Flash block
 - the FME file to use.

You can set up the memory map in a separate *Board/Chip Definition* (BCD) file, which you can then assign to your target connection. RealView Debugger provides FME files and BCD files for supported targets:

- FME files are located in the Flash examples directory
- BCD files for supported development boards and cores are located in your default setting directory:
`C:\Documents and Settings\userID\Local Settings\Application Data\ARM\rvdebug\version\shadowbase\etc`

If the FME files and BCD files provided do not meet your requirements, then you can create your own.

When you have configured your debug target to use the required BCD and FME files, you can program your Flash binary into the Flash device.

See also

- *Writing a binary to Flash* on page 6-4
- *Setting up a memory map* on page 9-15
- *Files in the default settings directory* on page B-3
- *Chapter 9 Mapping Target Memory*
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger* for details on creating an FME file, and creating, setting up, and assigning a BCD file to your debug target.

6.1.2 Flash Method files

FME files include code to:

- enable you to write to the Flash on your debug target
- perform read, write, and erase operations
- describe the way the Flash is configured on the bus.

Example files are included for all supported Flash devices as part of the root installation.

If you have a custom board or custom Flash type that is not one of those supported by RealView Debugger, you must create your own FME file.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger* for details on how to create an FME file.

6.1.3 Flash examples

The root installation contains a directory of examples for supported Flash devices. These examples contain the prebuilt FME files that are required to program the supported Flash devices.

All the Flash examples are located in:

install_directory\RVD\Flash\...\platform

Files are collected in subdirectories based on the target board or Flash device.

6.1.4 Basic procedure for programming a binary into Flash

The basic procedure for programming a binary into Flash, includes the following steps:

1. Configure a target connection to define a memory map that includes a Flash memory area.
 2. Connect to the target.
 3. Load the binary in preparation for writing to the Flash device.
 4. Write the binary to the Flash memory region that you have defined in the memory map.
- For example, the ARM® Integrator™/CP Flash starts at address 0x24000000.

Note

If you program the Flash on an ARM Integrator board using this release of RealView Debugger, you bypass the *ARM Firmware Suite* (AFS) Flash library system information blocks. These blocks are used by the AFS Flash Library, and are stored at the end of each binary written to Flash. If you rely on these blocks to keep track of what is in the Flash memory of your target, keep a record of the state and recreate it after working through the example.

See also

- *Connecting to a target* on page 3-27
- *Loading a binary* on page 4-12
- Chapter 9 *Mapping Target Memory*.

6.2 Writing a binary to Flash

Writing a binary to a Flash device requires the use of special algorithms, which are specific to each target. These algorithms are contained in the *Flash MMethod* (FME) files for each target.

The following procedure uses the ARM Integrator/CP board as an example of programming Flash:

- the FME file for this board is located in:

install_directory\RVD\Flash\...\platform\IntegratorCP

- the BCD file for this board is:

C:\Documents and Settings\userID\Local Settings\Application Data\ARM\rvdebug\version\shadowbase\etc\CP.bcd

The procedure assumes that the BCD file is already assigned to the target connection.

To write a binary to a Flash device:

- Make sure the binary is compiled to run in Flash.

For example, the Integrator/CP Flash starts at 0x24000000.

- Start RealView Debugger.

- Configure the required connection to provide a memory map of the target, which includes the Flash. For the Integrator/CP example, you can assign the CP.bcd file to the target connection.

- Connect to the target.

- Load the binary, and specify the location in Flash where you want to load (for example, 0x24000000).

The Flash device is opened, and RealView Debugger queues the binary in preparation for writing. The Flash Memory Control dialog box is displayed, shown in Figure 6-1.

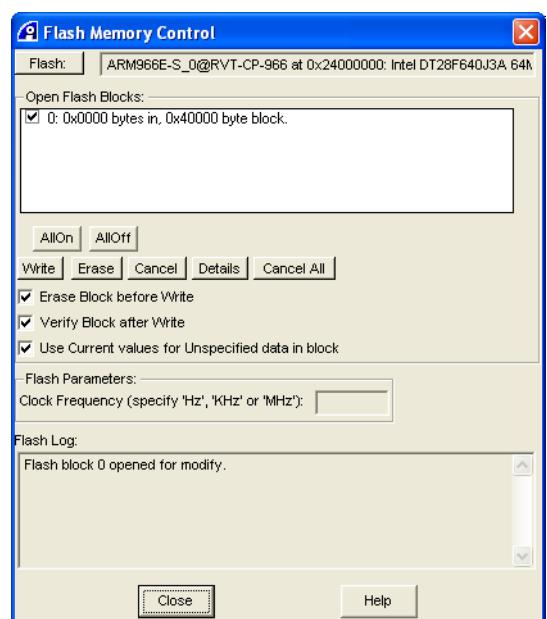


Figure 6-1 Flash Memory Control dialog box

Before you write the binary to Flash, do the following if required:

- view Flash details
 - choose the required operations.
6. The Clock Frequency field is enabled if it is required by your Flash device, and has the format $f[\text{Hz}|\text{kHz}|\text{MHz}]$. Enter the clock frequency f as a positive floating point number, for example, 14.175MHz. The unit specifier is optional, and defaults to Hz.
 7. Click **Write** to program the binary into Flash.
- Note**
- Wait for the write to complete before continuing.
8. Click **Close** to close the Flash Memory Control dialog box.
 9. Click the **Cmd** tab in the Output view to see the Flash operations.
 10. Select **Memory Map Tab** from the **View** menu to display the **Memory Map** tab where you can see the Flash memory area (green icon) on the Integrator/CP board. Expand the Flash memory map entry to see the Flash details.

Note

You can display the Flash Memory Control dialog box during your debugging session. To do this, select **Debug → Memory/Register Operations → Flash Memory Control...**, from the Code window main menu.

See also:

- *Considerations when writing a binary to Flash*
- *Connecting to a target* on page 3-27
- *Loading a binary* on page 4-12
- *Viewing information about the Flash memory* on page 6-8
- *Operations available when writing to Flash* on page 6-10
- *Setting up a memory map* on page 9-15
- *Chapter 9 Mapping Target Memory*.

6.2.1 Considerations when writing a binary to Flash

By default, RealView Debugger displays the Flash Memory Control dialog box in preparation for a write operation to Flash. You must manually perform the write operation using this dialog box. However, you can set up the Flash memory block so that the write operation is performed automatically.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger*
 - *Memory mapping Advanced_Information settings reference* on page A-20.

6.3 Writing to specific locations in Flash memory

Flash memory blocks are opened for access when you write to any location in Flash. Although you can load a binary to flash, you can also write to specific locations in Flash by using the Memory view.

To write to a specific location in Flash:

1. Connect to your target.
2. Select **Memory** from the **View** menu to display the Memory view, if it is not already visible.
3. Set the start address for the memory locations you want to view:
 - a. Right-click in the Memory view to display the context menu.
 - b. Select **Set Start Address...** from the context menu to display the Prompt dialog box.
 - c. Enter the required start address in Flash, for example, `0x24000000`.
 - d. Click **Set**. The Flash memory is displayed. The values are colored green to indicate that Flash memory is being displayed.
4. Right-click in the first byte of the location that you want to change.
5. Select **Set Value...** from the context menu.
6. Enter the new value at the prompt, for example `0xA0`.
7. Click **Set** to confirm this value.

The Flash device opens in preparation for writing, and the Flash Memory Control dialog box is displayed, shown in Figure 6-1 on page 6-4.

Before you have written to Flash, you can view the Flash details if required.

8. Make sure that the **Erase Block before Write** check box is selected.
9. If it is required by your Flash device, the Clock Frequency field on the Flash Memory Control dialog box is enabled, and has the format $f[\text{Hz}|\text{kHz}|\text{MHz}]$.
Enter the clock frequency f as a positive floating point number, for example, `14.175MHz`. The unit specifier is optional, and defaults to Hz.
10. Click **Write** to write to the chosen Flash location. Monitor the changes in the Memory view as memory is updated. The Flash Log confirms the Flash operation.
11. Click **Close** to close the Flash Memory Control dialog box.

See also

- *Considerations when writing to specific locations in Flash*
- *Viewing information about the Flash memory* on page 6-8
- *Setting memory with the Interactive Memory Settings dialog box* on page 14-15
- *Filling memory with a pattern* on page 14-16.

6.3.1 Considerations when writing to specific locations in Flash

By default, RealView Debugger displays the Flash Memory Control dialog box in preparation for a write operation to Flash. You must manually perform the write operation using this dialog box. However, you can set up the Flash memory block so that the write operation is performed automatically.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger*
 - *Memory mapping Advanced_Information settings* reference on page A-20.

6.4 Viewing information about the Flash memory

You can view details of Flash memory only when the Flash device is opened in preparation for writing to Flash, that is:

- when loading a binary
- when writing to specific locations in Flash.

Note

You cannot view Flash details after you have written the binary to Flash.

See also:

- *Viewing Flash information*
- *Viewing details on selected Flash blocks* on page 6-9

6.4.1 Viewing Flash information

To view Flash information:

1. Click **Flash**: in the Flash Memory Control dialog box.

The Flash Information Message dialog box is displayed, shown in Figure 6-2.

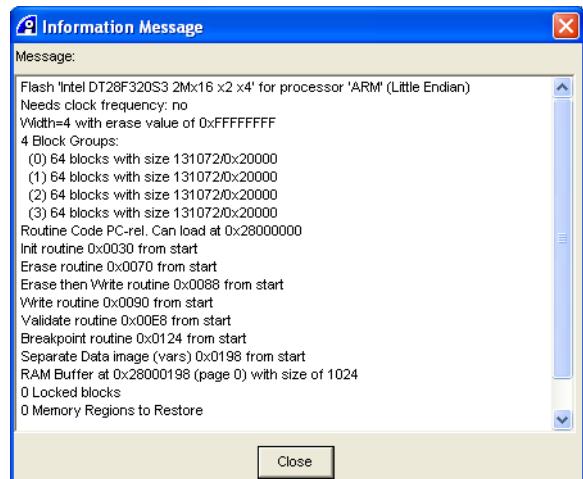


Figure 6-2 Flash information

2. Click **Close** to close the Information Message dialog box.

See also

- *Writing a binary to Flash* on page 6-4
- *Writing to specific locations in Flash memory* on page 6-6.

6.4.2 Viewing details on selected Flash blocks

To view details on selected Flash blocks:

1. Select the required Flash blocks in the Open Flash Blocks list of the Flash Memory Control dialog box. Figure 6-3 shows an example:

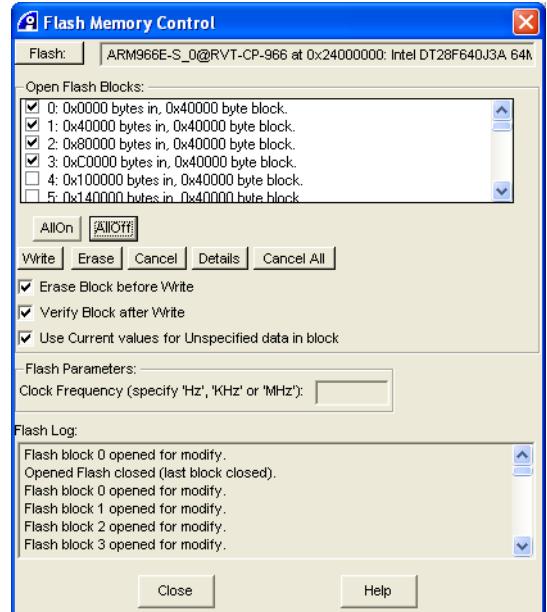


Figure 6-3 Flash Memory Control dialog box with multiple blocks selected

2. Click **Details** to display the Flash block Information Message dialog box. Figure 6-4 shows an example:

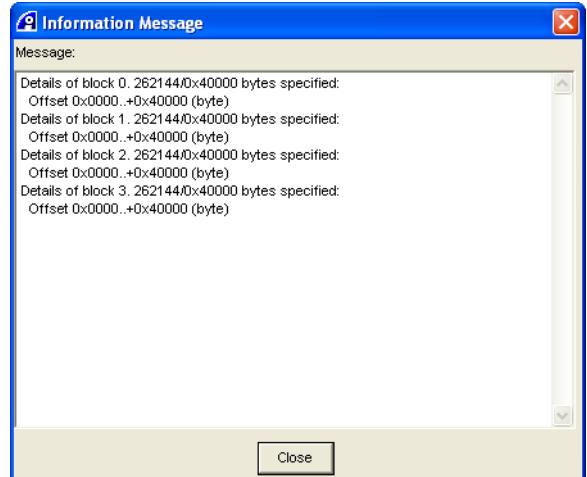


Figure 6-4 Flash block details

See also

- *Writing a binary to Flash* on page 6-4
- *Writing to specific locations in Flash memory* on page 6-6.

6.5 Operations available when writing to Flash

The Flash Memory Control dialog box (see Figure 6-1 on page 6-4) enables you to perform various operations when writing to Flash. You can perform these operations on one or more Flash blocks.

See also:

- *Selecting the Flash blocks to operate on*
- *Erasing selected Flash blocks*
- *Additional operations when writing to Flash*
- *Abandoning changes to Flash blocks* on page 6-11.

6.5.1 Selecting the Flash blocks to operate on

The Flash blocks are listed in the **Open Flash Blocks:** list of the Flash Memory Control dialog box (see Figure 6-1 on page 6-4). This lists the Flash blocks that are currently open for access. A check box is associated with each block. When this check box is selected, any Flash operations are performed on that block. When you first write a binary to Flash, only those Flash blocks that are affected by the binary are selected.

You can select the Flash blocks to operate on as follows:

- To select all Flash blocks, click **AllOn**. All blocks are selected by default.
- To select specific Flash blocks:
 1. Click **AllOff**.
 2. In the **Open Flash Blocks:** list, select the required Flash blocks.

6.5.2 Erasing selected Flash blocks

To erase selected Flash blocks using the Flash Memory Control dialog box (see Figure 6-1 on page 6-4), click **Erase**.

This normally sets every byte to 0x00 or 0xFF depending on the type of Flash being used.

6.5.3 Additional operations when writing to Flash

When you write to Flash blocks, you can also perform the following operations with the Flash Memory Control dialog box (see Figure 6-1 on page 6-4):

Erase Block before Write

Select this check box to erase the Flash block before performing the write operation.

This check box is selected by default.

Verify Block after Write

Select this check box to verify the Flash block, against the data source, after performing the write operation.

This check box is selected by default.

Use Current values for Unspecified data in block

Specifies that the original contents are to be maintained unless modified by the current operation. If deselected, the erase values are used.

This check box is selected by default. When selected, RealView Debugger preserves the current values in the unmodified parts of Flash when only a portion of a Flash block is modified. In this case, RealView Debugger first reads the unmodified portions of the flash block, and then writes the entire block, including the modified and unmodified portions. The read operation causes a small increase in programming time.

6.5.4 Abandoning changes to Flash blocks

You can abandon any changes you make to Flash blocks, using the following operations in the Flash Memory Control dialog box (see Figure 6-1 on page 6-4):

Cancel Abandons any changes made to the specified blocks of Flash.

Cancel All Abandons all changes to the Flash contents.

See also

- *Selecting the Flash blocks to operate on* on page 6-10.

Chapter 7

Debugging Multiprocessor Applications

This chapter describes in detail the features of RealView® Debugger that enable you to debug multiprocessor applications and compare the behavior of different targets, for example two ARM® processors.

This chapter contains the following sections:

- *About debugging multiprocessor applications* on page 7-2
- *Displaying multiple Code windows* on page 7-8
- *Attaching a Code window to a connection* on page 7-10
- *Unattaching a Code window from a connection* on page 7-11
- *Using DSTREAM or RealView ICE for multiprocessor debugging* on page 7-12
- *Synchronizing multiple processors* on page 7-13
- *Setting up software cross-triggering* on page 7-17
- *Setting up hardware cross-triggering* on page 7-20
- *Configuring embedded cross-triggering* on page 7-25
- *Configuring CoreSight embedded cross-triggering* on page 7-27
- *Sharing resources between multiple targets* on page 7-28.

7.1 About debugging multiprocessor applications

The following sections describe the features available in RealView Debugger that enable you to debug multiprocessor applications:

- *Working with multiple targets and connections*
- *Debug Interface units and multiple targets* on page 7-3
- *Synchronization and cross-triggering* on page 7-3
- *Synchronized start and stop operations, cross-triggering, and skid* on page 7-6
- *Synchronized stepping, cross-triggering, and skid* on page 7-6
- *Synchronized actions* on page 7-7.

7.1.1 Working with multiple targets and connections

RealView Debugger supports the debugging of multiprocessor applications in different ways, such as:

- Multiple simulator connections to:
 - *RealView ARMulator® ISS* (RVISS) targets (single-processor models)
 - *Instruction Set System Model* (ISSM) targets (single-processor models)
 - *Real-Time System Model* (RTSM) targets (single or multiprocessor models)
 - SoC Designer targets (single or multiprocessor models).
- Multiple hardware targets.
- Mixed hardware and simulated targets.
- A JTAG scan chain and a debug monitor.

By default, the last connection you make to a target is the *current connection*. However, if you have connections to multiple targets you can change the current connection if required.

RealView Debugger enables you to easily switch between connections without having to disconnect and reconnect the debugger.

When working with multiple connections, you might want to open multiple Code windows, and have the details for a different connection in each window. To do this, you must change the current connection and attach each Code window in turn to that connection. Only unattached Code windows show details of the current connection.

Target-specific connection settings

When you are working with multiple connections, you might want to use different connection settings for each target. For example, you might want to have semihosting enabled on one target, and disabled on all other targets.

Considerations when working with multiple connections

When working with multiple connections you can use a single Code window to view the connections, or set up multiple Code windows:

- If none of the processors performs semihosting operations, you can use a single, unattached Code window. You can cycle through the connections to see the debug state of each processor.

- You can use a single Code window if one processor performs semihosting operations. In this case, it is recommended that you attach the Code window to this processor. However, to view the state of any other processor, you might want to use at least one additional, unattached Code window. This enables you to cycle through the connections to see the debug state of each processor.
- If semihosting operations are to be performed by more than one processor connection then, for these connections at least, it is recommended that you:
 - use a separate Code window
 - attach each Code window to a different connection.

In this way, any application or debugger messages associated with the processor connections are displayed in the corresponding Code window.

See also

- *Changing the current target connection* on page 3-50
- *Attaching a Code window to a connection* on page 7-10
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Creating a target-specific Advanced_Information group* on page 3-23.

7.1.2 Debug Interface units and multiple targets

If all your hardware targets are on a single development platform, you can connect to those targets through a single hardware Debug Interface unit, such as DSTREAM or RealView ICE. This is the best configuration when performing multiprocessor debugging.

If real hardware is not available during your application development stage, you can connect to all simulated targets, or a mixture of hardware and simulated targets. For example, you can connect to two ARM processors through a RealView ICE Debug Interface unit, and connect to a simulated target through the RVISS Debug Interface. However, if you mix connections using a hardware Debug Interface unit and a simulator in this way, there are limitations on the multiprocessor debugging features you can use.

RealView Debugger supports multiple Debug Interface units by separating the target connection from your view of that connection.

Note

Limitations inherent in these different interfaces might have an impact on the way the debugger operates. For example, the speed of download might vary between interfaces or some features might not be available on some interfaces.

7.1.3 Synchronization and cross-triggering

When you have multiple processors that are cooperating within a single application, it is sometimes useful to control all processors with a single debugger command. RealView Debugger provides a synchronization mechanism to:

- synchronize execution start and stop
- synchronize single stepping
- synchronize specific actions.

Also, you might want to examine the state of all processors at one point in time. A processor stops for the following reasons:

- you told the debugger to stop it
- it activated a breakpoint

- it stopped because of cross-triggering
- the target operating environment stopped it.

Synchronization and cross-triggering differ as follows:

- Synchronization means that an operation initiated by the debugger is applied to all processors in the processor group. For example, clicking the **Stop** button stops all processors in the processor group.
- Cross-triggering means that, if CPU1 stops, CPU0 and CPU2 also stop.

You must manually identify the target connections that are to be synchronized or that are to take part in cross-triggering.

Note

Synchronization and cross-triggering are separate mechanisms. Therefore, processors do not have to be synchronized to take part in cross-triggering.

Definition of terms used in synchronization and cross-triggering

The following terms are used for synchronization and cross-triggering in RealView Debugger:

Processor group

In this section, the term *processor group* is used to refer to the set of processors that are configured to operate in a synchronized way.

You can synchronize:

- multiple simulated targets
- multiple hardware targets
- simulated targets with hardware targets.

Skid

For a processor group, *skid* is the time delay between the first processor stopping and the last processor stopping, and is an artefact of cross-triggering:

- When hardware cross-triggering is used, the delay involved (that is, the length of skid) might be a small number of clock cycles.
- When software cross-triggering is used, RealView Debugger must receive the stop request from the triggering target, process the request, and then send the stop request to each triggered target. Therefore, the skid time is far greater with software cross-triggering.

Loose synchronization

In some situations, RealView Debugger might synchronize processors loosely, such as in a mixed hardware and software environment or in a CoreSight system. This is characterized by a large skid, of as much as one to two seconds, because the synchronization is usually controlled by software. A large skid might also arise because there is no hardware synchronization control, so the debugger must issue start and stop commands individually.

When you add a target to the processor group for synchronized execution, and loose synchronization is in force, then RealView Debugger displays:

- a warning message in the Output view:
Synchronization will be loose (intrusive) for some or all connections in the synch group.
- a message in the Synchronization Control window:
Loosely coupled

Tight synchronization

In a hardware environment using a single DSTREAM or RealView-ICE unit, RealView Debugger uses a closely synchronized system where this is supported by the underlying processor or emulator. This has a very short skid, usually a few microseconds or less, and perhaps only a single instruction. This is because the synchronization controls are built into the target hardware, or hardware assistance is available in the DSTREAM or RealView ICE unit, or other JTAG emulator.

— Note —

Tight coupling is supported only for JTAG systems, that is ARM7, ARM9 and ARM11 processors. If there is a DAP, as in a CoreSight system, then tight coupling is not supported by DSTREAM or RealView ICE.

Cross-triggering

Cross-triggering occurs when one processor in a group stops because of an internal or an external event, which then causes other processors to stop.

The initial stop activates an external signal on the processor, for example **DBGACK**, that causes the cross-triggering hardware to generate an input signal to the other processors, for example **CPU0 stop**, that stops the processors. Each of these other processors skids before it stops.

For a target system that does not have hardware cross-triggering, the debugger can perform a similar function in software. However, the processes involved are more complex, and the skid time is much longer. For example, hardware cross-triggering might be able to stop all processors five target instructions after the initial breakpoint. A software solution might take many more target instructions.

If two processors, CPU0 and CPU1, are taking part in software cross-triggering and both reach an address that has a breakpoint, then there is a delay in processing the breakpoints:

1. The breakpoint on processor CPU1 is activated, which initiates a stop request to CPU0.
2. The software cross-triggering mechanism stops the processor CPU0. However, the breakpoint on CPU0 is not processed at this time.
3. When CPU1 restarts, it continues to run.
4. When CPU0 starts, the breakpoint on that processor is processed. CPU0 might stop or continue execution depending on the breakpoint actions.

See also

- *Synchronized actions* on page 7-7.

7.1.4 Synchronized start and stop operations, cross-triggering, and skid

Synchronization applies equally to starting processors and stopping them. Having a development platform with closely synchronized processors and a short skid enables you to stop the system and be confident that the overall state is as consistent as it was when you requested the stop. For a loosely synchronized system, whether the overall state is consistent when it has stopped is more dependent on the software and hardware architecture.

The length of skid varies and depends on many conditions:

- If one or more processors are controlled using debug monitor software, then the skid of that processor depends on whether the current task is interruptible or not.
- If one or more processors in the group share a memory bus, for example with a DMA controller, then another bus master can claim the bus and prevent the processor completing an instruction, so preventing it entering debug state.
- If the debugger must issue separate stop requests to each processor, then the host operating system might deschedule the debugger task between two of the stop requests and so introduce a significant extra delay.

Usually, a multiprocessor application is designed to be tolerant of differing execution speeds and differing execution orders for each of the constituent processes. In this case, communication attempts between processors are *guarded* to ensure data consistency. This is particularly true when the processors in a group run at differing clock speeds or using differing memory subsystems.

If communication guarding is not done, normal perturbations in the execution order might cause the application to fail. In communication systems that do not include very short communication timeouts, it is often possible to stop only one processor in a group. The other processors come to a halt through lack of, or excess of, data. Alternatively, you can let them continue to write to communication buffers while you work, intentionally overwriting them.

7.1.5 Synchronized stepping, cross-triggering, and skid

Synchronized execution applies to the stepping of processor groups, stopping them, and starting them. Having a target with closely synchronized processors enables you to step through the code to examine, for example, memory or registers on different processors. However, the behavior of each processor in the processor group during synchronized stepping operations depends on:

- the step operation being performed, which can be one of:
 - high-level step into
 - high-level step over
 - low-level step into
 - low-level step over
- whether synchronization is loosely or tightly coupled
- whether or not cross-triggering is in operation.

A synchronized low-level step into operation in the **Disassembly** tab is always predictable. However, because of skid, the other synchronized stepping operations might not behave in a predictable, or expected, way. These synchronized stepping operations can be unpredictable in the following circumstances:

- When function calls are involved on any of the processors.
- When any high-level step operation is performed in a source tab.

- When cross-triggering is in operation. In this case, the amount of skid depends on whether cross-triggering is controlled by hardware or by software (that is, the debugger):
 - For hardware cross-triggering, the skid is short. Also, the debugger does not have control over the cross-triggering process. As a result, if a processor stops because of a semihosting operation (including the use of a semihosting clock), the debugger has no control over the stopping of the other processors. When the processor restarts following the semihosting operation, then the debugger is unable to start the remaining processors.
 - For software cross-triggering, the skid is much longer because the debugger is controlling the cross-triggering process. Often this means that the synchronized step works as expected.

See also

- Execution control* on page 1-31.

7.1.6 Synchronized actions

You can specify one or more debugging actions to perform for synchronized processors. Table 7-1 shows the actions that can be performed, and the corresponding GUI operations and CLI commands.

Table 7-1 Synchronized actions and corresponding GUI operations

Action	GUI operation	CLI command
load image	Select Load Image... from the Target menu.	LOAD
unload image	Select Unload Image from the Target menu.	UNLOAD
reload image	Select Reload Image to Target from the Target menu.	RELOAD
restart	Select Set PC to Entry Point from the Debug menu.	RESTART
reset	Select Reset Target Processor from the Target menu.	RESET
set PC	Right-click in the left margin of the source or disassembly view, and select Set PC to Here from the context menu.	SETREG @PC= <i>address</i>
load binary	Select Load Binary... from the Target menu.	READFILE

Some of the GUI operations can also be performed from the Process Control view.

See also

- Image and binary loading* on page 1-26
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Alphabetical command reference* on page 2-12.

7.2 Displaying multiple Code windows

You might want to have more than one Code window displayed when you are debugging multiple targets, so that you can have the details for more than one connection visible at the same time.

With multiple connections established, you can create new windows to display different views during your debugging session.

See also:

- *Displaying a new Code window*
- *Properties of the new Code window*
- *What is window attachment?*.

7.2.1 Displaying a new Code window

To display a new Code window, select **New Code Window** from the **View** menu of any Code window.

If you frequently work with multiple Code windows, you do not have to close each Code window before you exit RealView Debugger. RealView Debugger remembers:

- the layout and size of each Code window
- the attachment state of each Code window
- the connection states.

If you close any Code window with the standard Windows Close button, or select **Close Window** from the **File** menu, then only that Code window closes. If you use these methods to close the last Code window, then RealView Debugger exits. The state of any Code window that you previously closed is not saved.

7.2.2 Properties of the new Code window

The new Code window has the following properties:

- it has the default layout and window size
- the window attachment is the same as the Code window you used to open the new Code window
- the connection details displayed depend on the window attachment:
 - if unattached, the details of the current connection are displayed
 - if attached, the details of the connection to which the window is attached are displayed.

See also

- *What is the current connection?* on page 3-50
- *What is window attachment?*
- *Attaching a Code window to a connection* on page 7-10.

7.2.3 What is window attachment?

With a connection established, *attachment* refers to the state where a Code window is tied to a particular connection. Therefore, if a Code window is:

Attached It displays information only for the attached connection.

Unattached It displays information only for the current connection.

You can use a combination of attached and unattached Code windows as required.

— Note —

You can attach more than one Code window to the same connection. This might be useful when you are debugging OS-aware connections.

See also

- *RealView Debugger RTOS Guide*.

7.3 Attaching a Code window to a connection

If you have multiple connections and multiple Code windows, then by default only the information for one connection, the current connection, is visible in all your Code windows. If you want to view the details of more than one connection at the same time, then you must use the window attachment feature of RealView Debugger.

To attach a Code window to a connection:

1. Make sure the Code window is not currently attached to a connection.
 2. Make sure the details of the required connection are shown in the Code window that you want to attach to that connection.
-  To do this, click **Cycle Connections** to make the connection of interest the current connection.
3. To attach the Code window to the chosen connection:
 - a. Click the drop-down arrow on the **Cycle Connections** button to display the active connections menu.
 - b. Select **Attach Window to a Connection** from the menu.

The attachment state [Target] is added to the Code window title bar immediately after the connection name, for example:

ARM7TDMI@RVISS [Target] - RealView Debugger

In the active connections menu, shown in Figure 7-1:

- The connection that is attached to the Code window is marked with a check mark. The list of active connections does not show the attached state of other Code windows, only the one that you are currently using.
- A connection that is marked with an asterisk indicates that it is the current connection.

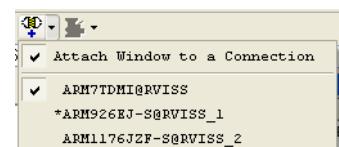


Figure 7-1 Active connections list showing attached connection

See also:

- *What is window attachment?* on page 7-8
- *Changing the current target connection* on page 3-50.

7.4 Unattaching a Code window from a connection

You can unattach a Code window from a connection, if required.

To unattach a Code window from a connection:

1. Move the focus to the Code window that is to be unattached.

2. To unattach the Code window from the connection:



a. Click the drop-down arrow on the **Cycle Connections** button to display the active connections menu.

b. Select **Attach Window to a Connection** from the menu.

The Code window shows the connections details of the current connection.

The check mark is removed from both the **Attach Window to a Connection** option, and the corresponding connection option.

The title bar of the Code window is updated to show that it is unattached, for example:

ARM1176JZF-S@RVISS_2 - RealView Debugger

You can cycle through the connections to view the details of another connection.

See also

- *What is window attachment?* on page 7-8
- *Changing the current target connection* on page 3-50.

7.5 Using DSTREAM or RealView ICE for multiprocessor debugging

A single DSTREAM or RealView ICE unit enables you to debug multiple targets.

Depending on your target hardware configuration, you might have to use more than one DSTREAM or RealView ICE Debug Interface unit. If this is the case, you must create a separate Debug Configuration for each Debug Interface unit.

See also:

- *About creating a Debug Configuration* on page 3-8.
- *Debug Interface units and multiple targets* on page 7-3.

7.6 Synchronizing multiple processors

You synchronize processors in a multiprocessor system when you want to perform specific actions on all the processors at the same time, such as stepping.

See also:

- *Synchronization and cross-triggering* on page 7-3
- *Synchronization controls*
- *Synchronizing actions*
- *Synchronizing execution operations* on page 7-15
- *Considerations when synchronizing multiple processors* on page 7-16.

7.6.1 Synchronization controls

The synchronization controls are available on the Synchronization Control window, shown in Figure 7-2. If any connections are established, then the processors for those connections are listed.

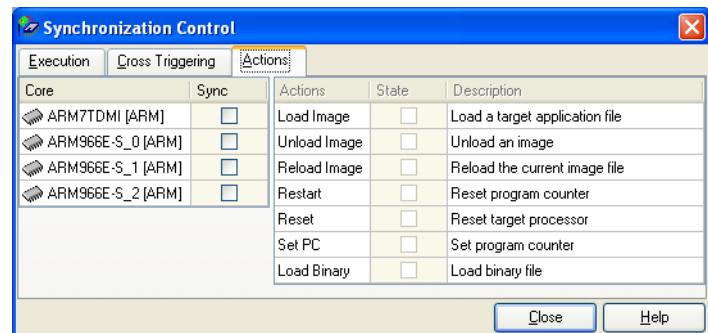


Figure 7-2 The Synchronization Controls

Synchronization groups

The controls are divided into the following synchronization groups, which can be set up independently:

- Actions** This group shows the processors that are synchronized for actions, and the actions that are to be performed.
- Execution** This group shows the processors that are synchronized for execution operations, and the execution operations that are to be performed.

7.6.2 Synchronizing actions

To synchronize the processors for specific actions:

1. Select **Synchronization Control...** from the **Target** menu to display the Synchronization Control window.
2. Click the **Actions** tab to display the action controls. Figure 7-3 on page 7-14 shows an example:

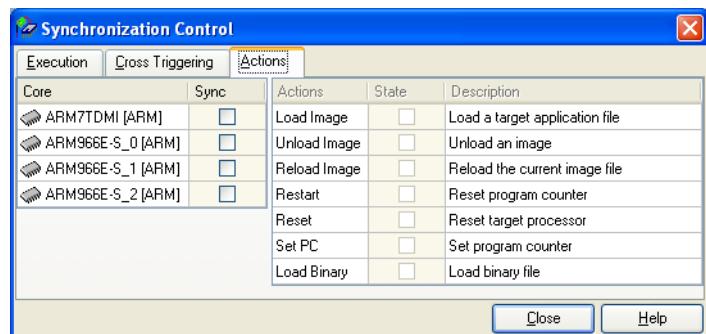


Figure 7-3 Synchronization Control window

3. Select the **Sync** check box associated with each processor that you want to include in the processor group.
4. Deselect the check box associated with each action that you do not want to be synchronized. The actions have the following meanings:

Load Image

A specified image is loaded onto all synchronized processors.

Unload Image

The image on each synchronized processor is unloaded.

Reload Image

The image on each synchronized processor is reloaded.

Restart Reset the program counter to the image entry point for all synchronized processors.

Reset All synchronized processors are reset.

Set PC Set the program counter to a single value on all synchronized processors.

Load Binary

A specified binary file is loaded onto all synchronized processors.

Figure 7-4 shows an example Synchronization Control window with a processor group containing the synchronized processors ARM966E-S_0, ARM966E-S_1, and ARM966E-S_2.

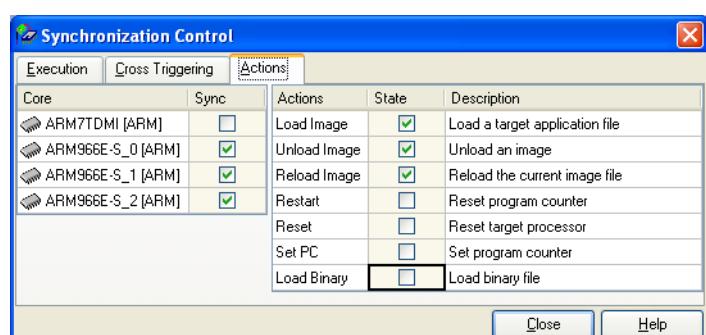


Figure 7-4 Synchronization Control window showing synchronized actions

See also

- *Synchronized actions* on page 7-7.
- *Synchronization controls* on page 7-13.

7.6.3 Synchronizing execution operations

To synchronize the processors for specific execution operations:

1. Select **Synchronization Control...** from the **Target** menu to display the Synchronization Control window.
2. Click the **Execution** tab to display the execution controls. Figure 7-5 shows an example:

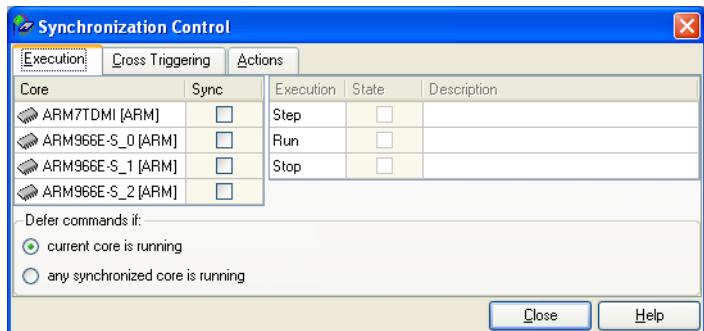


Figure 7-5 Execution controls in the Synchronization Control window

3. To synchronize specific execution operations:
 - a. Click the **Sync** check box associated with each processor to be synchronized for execution operations.
 - b. Deselect the check box associated with each execution operation that you do not want to be synchronized. The operations have the following meanings:
 - Step** The processor group is synchronized on stepping operations. That is, if you step one processor in the group, then all other processors in the group are stepped.
 - Run** The processor group is synchronized on run instructions. That is, if you start one processor in the group, then all other processors in the group are started.
 - Stop** The processor group is synchronized on stop instructions. That is, if you stop one processor in the group, then all other processors in the group are stopped.
 - c. Select the required Command Pending Mode to determine the behavior when a CLI command is issued that can be pended:

current core running

The command is pended only if the processor shown in the Code window is running. This is the default.

Note

This might lead to unpredictable behavior, particularly when using scripts.

any synchronized core is running

The command is pended if any processor synchronized to the current processor is running.

Figure 7-6 on page 7-16 shows an example Synchronization Control window containing the synchronized processors ARM966E-S_0, ARM966E-S_1, and ARM966E-S_2. In this example, the Execution controls show that both processors are to start together when you start one processor. However, if you click the **Stop** button, then only the processor for the connection shown in the current Code window stops. The processor ARM7TDMI is not affected.

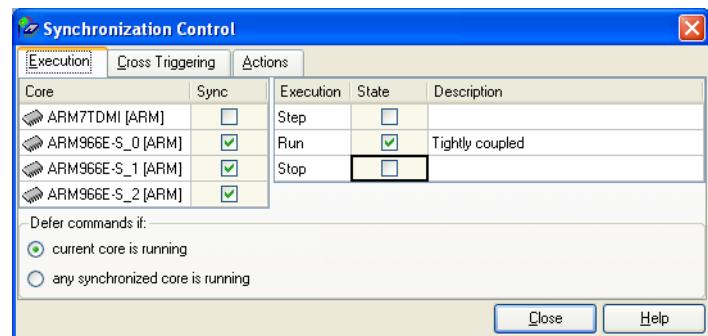


Figure 7-6 Synchronization Control window showing synchronized execution operations

See also

- *Code window status bar* on page 1-5
- *Synchronization controls* on page 7-13
- *Synchronized stepping, cross-triggering, and skid* on page 7-6.

7.6.4 Considerations when synchronizing multiple processors

Be aware of the following when synchronizing multiple processors:

- Using synchronized stepping with cross-triggering is not recommended, because the results can be unpredictable.
- The run state in the Code window status bar indicates Waiting when a stopped processor is in the middle of a synchronized multiple step operation, and it is waiting for another processor to stop before it continues with the multiple step.

7.7 Setting up software cross-triggering

You set up cross-triggering for processors in a multiprocessor system when you want one processor to affect the behavior of other processors, without user intervention. For example, you might want all processors to stop when a breakpoint is activated.

The software cross-triggering controls are available for all connected processors, including processors that support hardware cross-triggering. You can also use the hardware cross-triggering controls if required, or have mixed hardware and software cross-triggering.

Note

If your hardware supports more complex cross-triggering controls, you must set these manually. Alternatively, you can create a script to set the appropriate configuration. For example, if your hardware supports the Embedded Cross Trigger unit, see:

- *Configuring embedded cross-triggering* on page 7-25
- *Configuring CoreSight embedded cross-triggering* on page 7-27.

See also:

- *Synchronization and cross-triggering* on page 7-3
- *Synchronization controls* on page 7-13
- *Synchronizing actions* on page 7-13
- *Synchronizing execution operations* on page 7-15
- *Considerations when synchronizing multiple processors* on page 7-16.

7.7.1 Cross-triggering controls

The cross-triggering controls are available on the **Cross Triggering** tab of the Synchronization Control window, shown in Figure 7-7. These controls describe the communications between two or more processors.

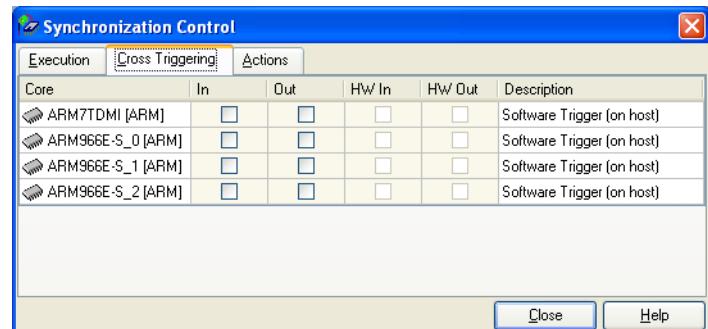


Figure 7-7 Synchronization Control window showing cross-triggering controls

If any connections are established, then the processors for those connections are listed. If you make a connection after opening the Synchronization Control window, then the processor for the new connection is added to the list of processors. However, the related processor and cross-triggering controls are not selected.

To set up software cross-triggering:

1. Select **Synchronization Control...** from the **Target** menu to display the Synchronization Control window. Figure 7-8 on page 7-18 shows an example:

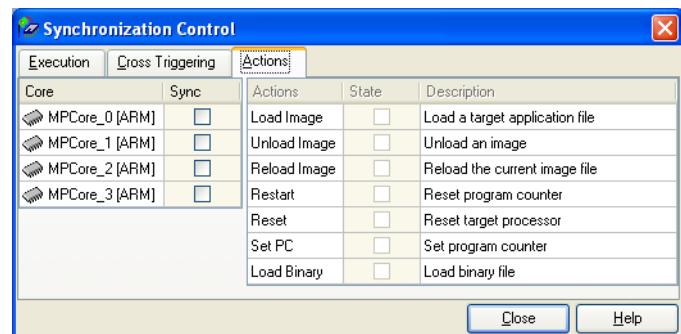


Figure 7-8 Synchronization Control window

- Click the Cross Triggering tab. A row of cross-triggering controls is available for each currently connected processor. Figure 7-9 shows an example:

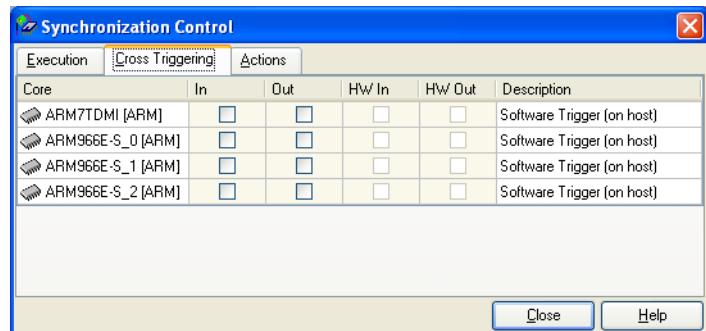


Figure 7-9 Cross Triggering tab in the Synchronization Control window

- Decide which processors are to take part in cross-triggering. Also, decide which processor is to have the Out trigger set, that is, the one that is to have control over the other processors. The remaining processors that you want to use for cross-triggering have the In trigger set.

————— Note —————

More than one processor can have the Out trigger set, or both the Out and In triggers set.

- Select the **In** and **Out** check boxes as required to set up the trigger relationships:
- In** The processor is to respond to the stop request of a processor that has the **Out** check box selected.
- Out** When the processor stops, it is to broadcast a stop request to processors that have the **In** check box selected.

In the example shown in Figure 7-10 on page 7-19, when the ARM966E-S_0 processor stops, then the ARM966E-S_1 and ARM966E-S_2 processors stop. The ARM7TDMI processor is not affected by the cross-triggering of the other processors.

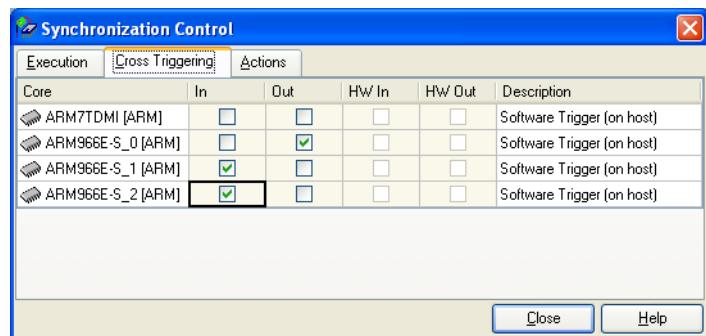


Figure 7-10 Cross-triggering controls

5. Load the required images, and set up the debugging conditions for your targets. For example, set any breakpoints.
6. You might want to start the processors with the In trigger enabled first (the ARM966E-S_1 and ARM966E-S_2 processors). Otherwise, the processor with the Out trigger enabled (the ARM966E-S_0 processor) might stop before you have time to start the other processors. For example, you might have set one or more breakpoints.

You can either start each processor manually, or specify a connection sequence in the Debug Configuration.

See also

- *Considerations when using software cross-triggering*
- *Cross-triggering controls* on page 7-17
- *Setting up hardware cross-triggering* on page 7-20
- *Configuring embedded cross-triggering* on page 7-25
- *Configuring CoreSight embedded cross-triggering* on page 7-27
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring a connection sequence for multiple targets* on page 3-36.

7.7.2 Considerations when using software cross-triggering

Be aware of the following when using software cross-triggering:

- Using synchronized stepping with cross-triggering is not recommended, because the results can be unpredictable.

7.8 Setting up hardware cross-triggering

If supported by your target processors (such as an ARM11™ MPCore™), you can set up hardware cross-triggering. For these processors, the HW Out and HW In controls are available.

— Note —

The procedure described in the following section does not apply to CoreSight embedded cross-triggering.

Hardware cross-triggering requires that you first set up the appropriate memory mapped registers and the CLI commands required to manipulate those registers. Typically, these are set up in a processor-specific BCD file, which you assign to your Debug Configuration:

- If you are debugging on an ARM architecture-based processor, then a BCD file might already exist with the appropriate CLI commands defined.
- Otherwise, you must create your own BCD file.

RealView Debugger automatically runs the cross-triggering CLI commands set up in the BCD file when you enable and disable the HW In and HW Out triggers.

— Note —

If your hardware supports more complex cross-triggering controls, you must set these manually. Alternatively, you can create a script to set the appropriate configuration.

See also:

- *Setting up the hardware cross-triggering controls* on page 7-21
- *Reviewing the register definitions and CLI commands for the ARM11 MPCore* on page 7-23.

7.8.1 Setting up the hardware cross-triggering controls

To set up the hardware cross-triggering controls:

1. Connect to the target processors on your development platform.

————— Note —————

You must display the **Configuration** grouping of your targets to be able to connect to all targets on a development platform in a single operation.

2. Select **Synchronization Control...** from the **Target** menu to display the Synchronization Control window. Figure 7-11 shows an example:

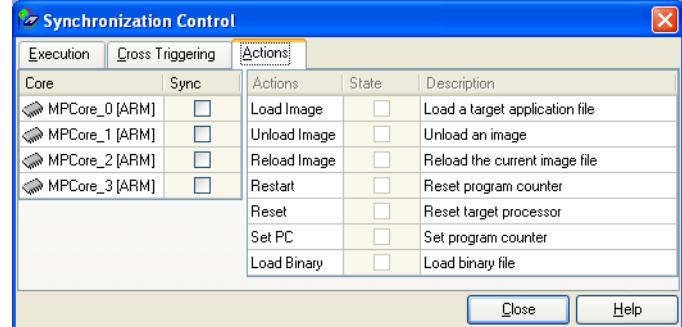


Figure 7-11 Synchronization Control window

3. Click the **Cross Triggering** tab. A row of cross triggering controls is available for each currently connected processor. Figure 7-12 shows an example:

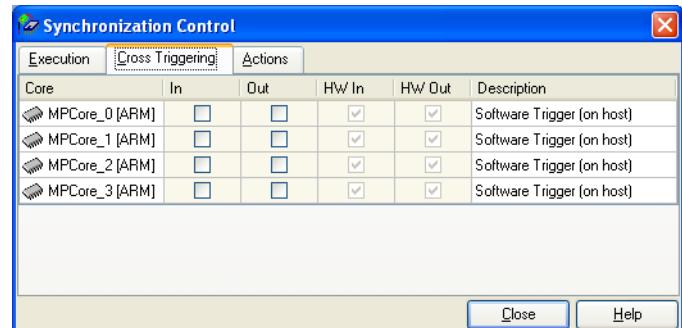
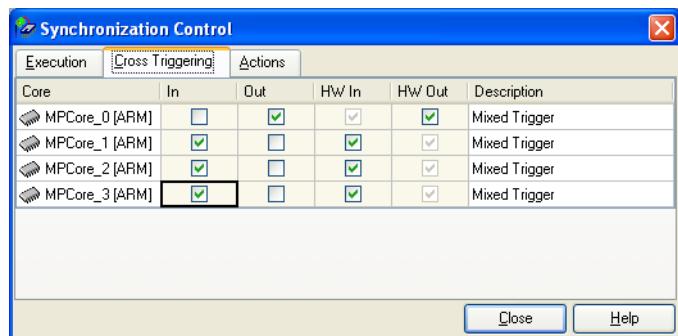


Figure 7-12 Hardware cross-triggering controls

The hardware cross trigger check boxes **HW In** and **HW Out** are selected to indicate that hardware cross-triggering is available. However, the check boxes are disabled by default.

4. For each processor that is to take part in hardware cross-triggering, select the **In** or **Out** check box as required to enable the corresponding **HW Out** and **HW In** check boxes. Figure 7-13 on page 7-22 shows an example:

**Figure 7-13 Hardware cross-trigger controls enabled**

The hardware cross-trigger relationships are as follows:

HW In The processor is to respond to the stop request of a processor that has the **HW Out** check box enabled.

HW Out When the processor stops, it is to broadcast a stop request to processors that have the **HW In** check box enabled.

In the example shown in Figure 7-13, when the MPCore_0 processor stops, then the MPCore_1, MPCore_2, and MPCore_3 processors stop.

If you want a processor to use software cross-triggering for either or both triggers, then:

- a. Select the **In** and **Out** check boxes as required.

- b. Deselect the corresponding **HW In** and **HW Out** check boxes.

5. Load the required images, and set up the debugging conditions for your targets. For example, set any breakpoints.
6. You might want to start the processors with the **HW In** trigger enabled first (the MPCore_1, MPCore_2, and MPCore_3 processors). Otherwise, the processor with the **Out** trigger enabled (the MPCore_0 processor) might stop before you have time to start the other processors. For example, you might have set one or more breakpoints.

You can either start each processor manually, or specify a connection sequence in the Debug Configuration.

Note

Be aware that using synchronized stepping with cross-triggering is not recommended, because the results can be unpredictable.

See also

- *Connecting to multiple targets* on page 3-46
- *Connecting to all targets for a specific Debug Configuration* on page 3-48
- *Cross-triggering controls* on page 7-17
- *Setting up software cross-triggering* on page 7-17
- *Configuring embedded cross-triggering* on page 7-25
- *Configuring CoreSight embedded cross-triggering* on page 7-27
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring the CLI commands for hardware cross-triggering* on page 3-33
 - *Configuring a connection sequence for multiple targets* on page 3-36
 - *Creating a custom memory mapped register* on page 4-37.

7.8.2 Reviewing the register definitions and CLI commands for the ARM11 MPCore

To review the memory mapped register definitions and the CLI commands for the ARM11 MPCore:

1. Select **Connect to Target** from the **Target** menu. The Connect to Target window is displayed.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Debug Interface containing the Debug Configuration to be customized.
4. Make sure that there are no targets connected on the Debug Configuration.

————— **Note** —————

You cannot customize a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

5. Right-click on the required Debug Configuration to display the context menu.
6. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
7. Click the **BCD files** tab to display the available definitions and assignments.
8. Locate the ARM11MP definition and select it.
9. Click the **View Definition...** button below the list containing the definition.

The Connection Properties window is displayed with the ARM11MP group selected.

Figure 7-14 shows an example:

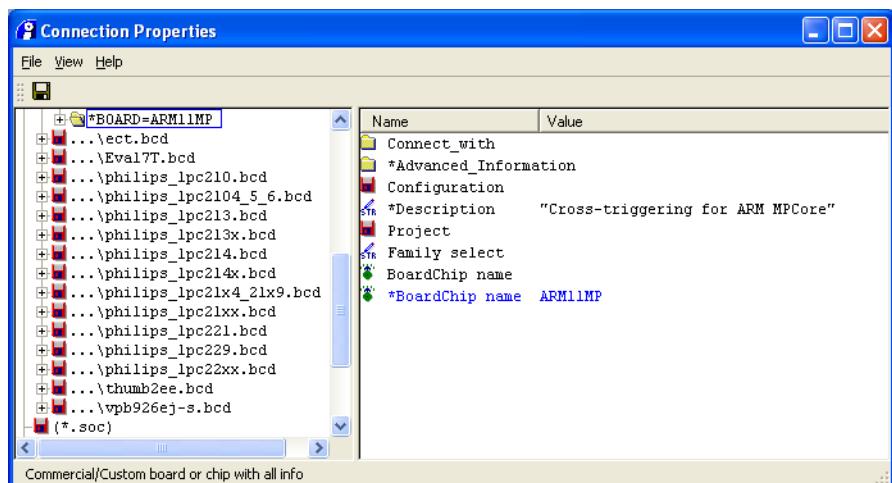


Figure 7-14 Connection Properties window

10. To review the register definitions and CLI commands:
 - a. Right-click on the BOARD=ARM11MP group in the left pane to display the context menu.
 - b. Select **Expand whole Tree** from the context menu.
 - c. Select the *Cross_trigger group in the left pane. The CLI commands to enable and disable the in and out triggers are shown in the right pane.
 - d. Select the *Register group in the left pane. The memory mapped registers used in the cross-triggering CLI commands are shown.
 - e. Select each register in the left pane to view the register settings.

11. Select **Close Window** from the **File** menu to close the Connection Properties window.
12. Click the **OK** button to close the Connection Properties dialog box.

See also

- *Connecting to multiple targets* on page 3-46
- *Connecting to all targets for a specific Debug Configuration* on page 3-48
- *Cross-triggering controls* on page 7-17
- *Setting up software cross-triggering* on page 7-17
- *Configuring embedded cross-triggering* on page 7-25
- *Configuring CoreSight embedded cross-triggering* on page 7-27
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring the CLI commands for hardware cross-triggering* on page 3-33
 - *Configuring a connection sequence for multiple targets* on page 3-36
 - *Creating a custom memory mapped register* on page 4-37.

7.9 Configuring embedded cross-triggering

An *Embedded Cross Trigger* (ECT) unit provides a mechanism for passing debug events between multiple processors, and allows synchronized debug and trace of an entire system on chip. If your target hardware supports an ECT unit, then you can use the ECT.bcd file provided with RealView Debugger to configure this logic for each target on your development platform. The following procedure describes how to set up the ECT for processor cross-triggering. For tracing, triggers are set up using tracepoints.

— Note —

If your target hardware supports CoreSight™ ECT, see *Configuring CoreSight embedded cross-triggering* on page 7-27.

— Note —

Be aware that using synchronized stepping with cross-triggering is not recommended, because the results can be unpredictable.

To configure an ECT unit:

1. Select **Connect to Target** from the **Target** menu. The Connect to Target window is displayed.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Debug Interface containing the Debug Configuration to be customized.
4. Make sure that there are no targets connected on the Debug Configuration.

— Note —

You cannot customize a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

5. Right-click on the required Debug Configuration to display the context menu.
6. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
7. Click the **BCD files** tab to display the available definitions and assignments.
8. To assign an ECT board/chip definition to this configuration:
 - a. Select **ECT** in the Available Definitions list.
 - b. Click the **Add** button.
 The ECT definition is moved from the Available Definitions list to the Assigned Definitions list.
9. Select **OK** to save your changes.

When you connect to the target, an **ECT** tab is provided in the Registers view. Figure 7-15 on page 7-26 shows an example:

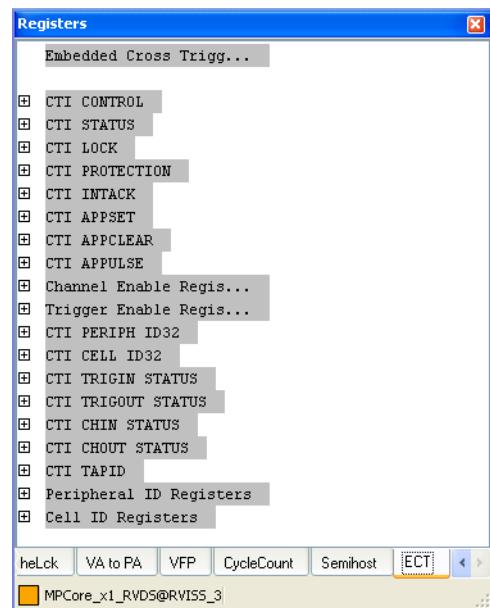


Figure 7-15 ECT tab

See also

- *Setting up software cross-triggering* on page 7-17
- *Setting up hardware cross-triggering* on page 7-20
- *Configuring CoreSight embedded cross-triggering* on page 7-27
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*
- the following in the *RealView Debugger Trace User Guide*:
 - Chapter 5 *Tracepoints in RealView Debugger*
- *Embedded Cross Trigger Technical Reference Manual*.

7.10 Configuring CoreSight embedded cross-triggering

The CoreSight ECT provides a mechanism for passing debug events between multiple processors, and allows synchronized debug and trace of an entire system on chip. The following procedure describes how to set up the *Cross Trigger Interface* (CTI) of the CoreSight ECT for processor cross-triggering. For tracing, triggers are set up using tracepoints.

— Note —

Be aware that using synchronized stepping with cross-triggering is not recommended, because the results can be unpredictable.

To configure a CoreSight CTI for embedded cross-triggering of multiple processors:

1. Make sure you have added all the CoreSight components available on your development platform to the DSTREAM or RealView ICE Debug Configuration you are using:
 - a. Select **Connect to Target...** from the **Target** menu to display the Connect to Target window.
 - b. Expand the required Debug Interface, for example RealViewICE.
 - c. Expand the Debug Configuration you are using.
2. Connect to the CoreSight CTI (the CSCTI_n target) that you want to configure.

When you connect to the CoreSight CTI, a **Device** tab is provided in the Registers view. Figure 7-16 shows an example:

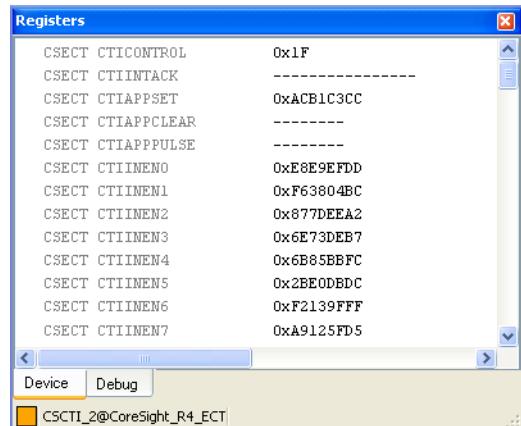


Figure 7-16 CoreSight CTI registers in the Registers view

See also

- *Configuring embedded cross-triggering* on page 7-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - *About customizing a DSTREAM or RealView ICE Debug Interface configuration* on page 2-3
- the following in the *RealView Debugger Trace User Guide*:
 - *Chapter 5 Tracepoints in RealView Debugger*
- *CoreSight Components Technical Reference Manual*.

7.11 Sharing resources between multiple targets

The following sections describe how RealView Debugger is affected by multiprocessor debugging:

- *Resource sharing and debugger consistency*
- *Saving and restoring your .brd file*
- *Locating the settings for shared memory regions* on page 7-29
- *Defining memory for a symmetric multiprocessor environment* on page 7-30
- *Defining memory for a three processor multimedia development platform* on page 7-31
- *Shared program memory* on page 7-35.

7.11.1 Resource sharing and debugger consistency

RealView Debugger provides a mechanism whereby changes to a given connection cause an update for each related connection. It does this by enabling you to define shared memory areas for different processors as part of your target configuration settings.

Be aware that changes to the shared memory region are updated only for the memory view of the target that makes the change. If you have other Code windows open, then to see the changes in each Code window you must manually update the memory view in those Code windows.

See also

- *Locating the settings for shared memory regions* on page 7-29
- *Manually updating values in the Memory view* on page 13-54.

7.11.2 Saving and restoring your .brd file

In these examples you are amending your board file. By default, the board file information is held in rvdebug.brd, which is normally stored in your RealView Debugger home directory.

Other configuration files have extensions such as *.auc, *.smc, and *.rvc.

You are recommended to backup your home directory before starting the examples described in this chapter, so that you can restore your original configuration later. If you have to restore your board file, exit RealView Debugger, and restore it from the backup directory.

You can restart RealView Debugger with your saved configuration.

Returning to the installation settings

If you want to return to the installation settings:

1. Exit RealView Debugger.
2. Delete or move your RealView Debugger home directory.
3. Restart RealView Debugger. RealView Debugger creates a new default configuration for you.

See also

- *Defining the home directory on Windows* on page 2-10
- *The home directory on Red Hat Linux* on page 2-10.

7.11.3 Locating the settings for shared memory regions

The settings that enable you to define shared memory regions are available from the Connection Properties window where you configure the settings for each processor that is sharing memory. The settings are in the **Memory_block** group of an **Advanced_Information** block.

To locate the settings for defining shared memory regions:

1. Display the Connect to Target window.
2. Right-click on the Debug Configuration containing the processors that are to share the memory to display the context menu.

— Note —

Only processors within the same Debug Configuration can access same shared memory region.

3. Locate the settings for defining shared memory regions:
 - a. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
 - b. Click the **BCD files** tab to display the available definitions and assignments.
 - c. In the Assigned Definitions list, select the board/chip definition that contains the settings for the shared memory regions.
 - d. Click the related **View Definitions...** button.

The Connection Properties window is displayed. The selected Debug Configuration is highlighted in the left pane and the contents of this entry are in the right pane.
4. Expand the following groups in turn:
 - a. **BOARD=name**
 - b. **Advanced_Information**.
 - c. Default, or connection-specific name.
 - d. **Memory_block**.
5. Double-click on the Default entry in the left pane to expand it. Figure 7-17 shows an example:

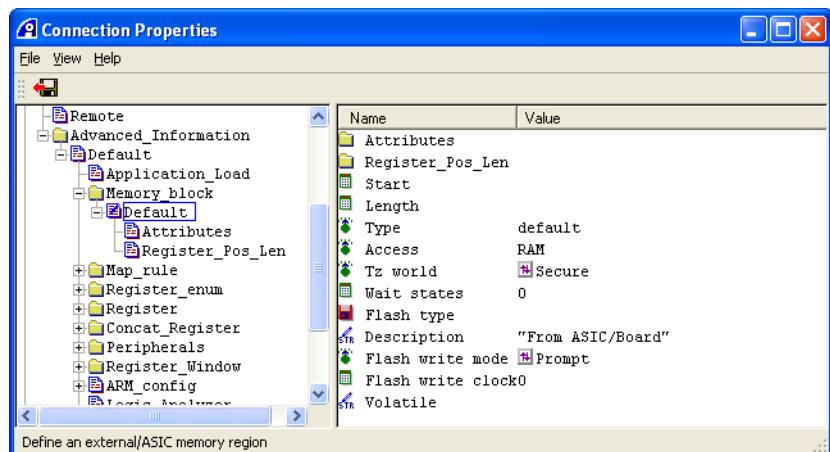


Figure 7-17 Defining shared memory regions

You use the **Memory_block** group to define areas of memory that have specific characteristics, one of which is whether the memory region is to be shared between multiple processors.

6. Rename the Default memory block:
 - a. Right-click on Default in the left pane to display the context menu.
 - b. Select **Rename** from the context menu to display a create dialog box.
 - c. Enter **SharedRAM**.
 - d. Click **Rename**.
7. Select the Attributes group in the left pane to see the settings Shared and Shared_id.

See also

- *Defining memory for a symmetric multiprocessor environment*
- *Defining memory for a three processor multimedia development platform* on page 7-31
- *Shared program memory* on page 7-35
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

7.11.4 Defining memory for a symmetric multiprocessor environment

A simple example is a SMP environment, where two processors share all memory and all peripherals. To do this, change the settings for each processor (shown in the example in Figure 7-17 on page 7-29):

1. Locate the **Memory_block** group for the connection.
2. Expand the **Memory_block** group.
3. Select **SharedRAM** in the left pane.
4. Define the region for the memory block:
 - a. Set the value of **Start** to the start address for this memory block, for example **0x10000**.
 - b. Set the value of **Length** in memory units for the memory block. For example, **0xFFFF** bytes for an ARM architecture-based processor.
5. Expand the **SharedRAM** group.
6. Select **Attributes** in the left pane to display the list of attributes for this memory block. Figure 7-18 shows an example:

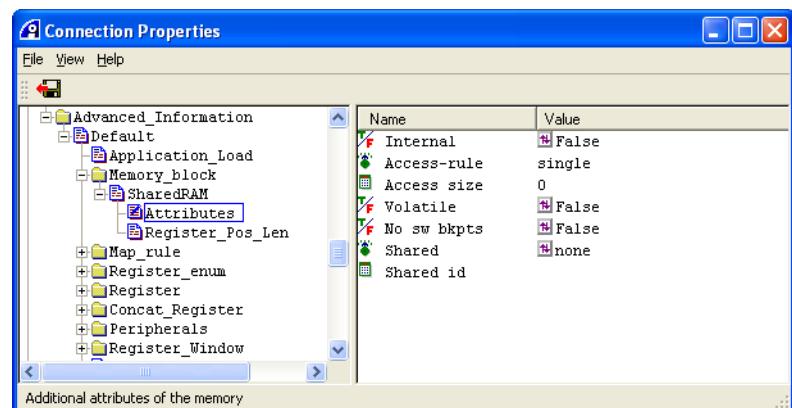


Figure 7-18 Memory block attributes

7. Set up the attributes for the memory block:
 - a. Set the value of Shared to **direct**.
 - b. Set the value of Shared_id to **0x1**.

You can use any integer value for the share ID in the range 0 to 65535. However, you must use the same value for each block that is sharing the same memory area. This enables RealView Debugger to identify which memory blocks relate to the same shared memory area.

8. If this is a code region for both processors, then it is not advisable to use software breakpoints. To prevent software breakpoints being set in this memory area, set the No_sw_bkpts setting to **True** (see Figure 7-18 on page 7-30).

See also

- *Locating the settings for shared memory regions* on page 7-29
- *Software breakpoints* on page 11-3.

7.11.5 Defining memory for a three processor multimedia development platform

A more complex example configuration, shown in Figure 7-19 on page 7-32, shows the address spaces of three processors sharing two memory regions.

Each entry in the Advanced_Information section of the board file describes the memory layout of a processor as one or more segments. For each processor and for each segment, the board file must include:

- the base memory address and length
- the type of memory, that is read-only, or read/write
- whether the segment is shared, and if so the share ID.

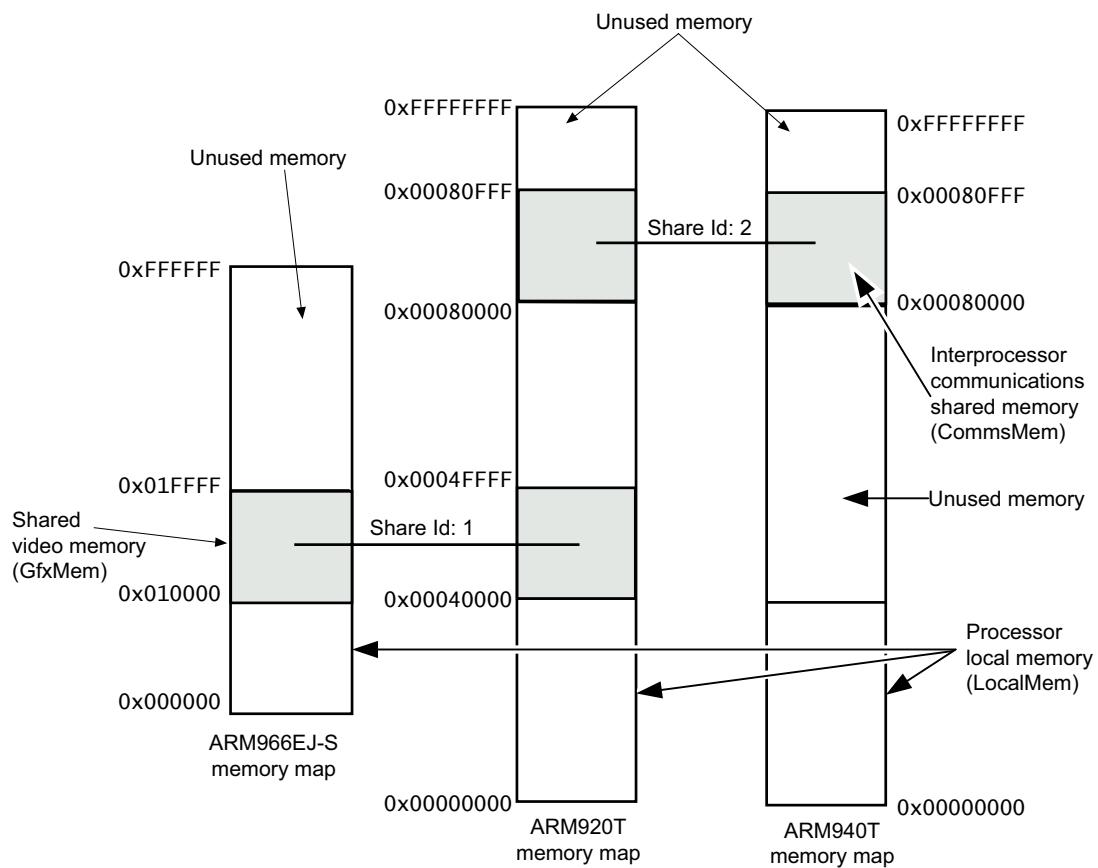


Figure 7-19 Example of a shared memory configuration

The details of the settings that you must make in your Connection Properties window to configure the three processors are shown in Figure 7-19. You must start by creating a Board/Chip definition file, similar to the CP.bcd shown in Figure 7-20, and then referencing it from the board file by using the `BoardChip_name` setting in the `CONNECTION=` group.

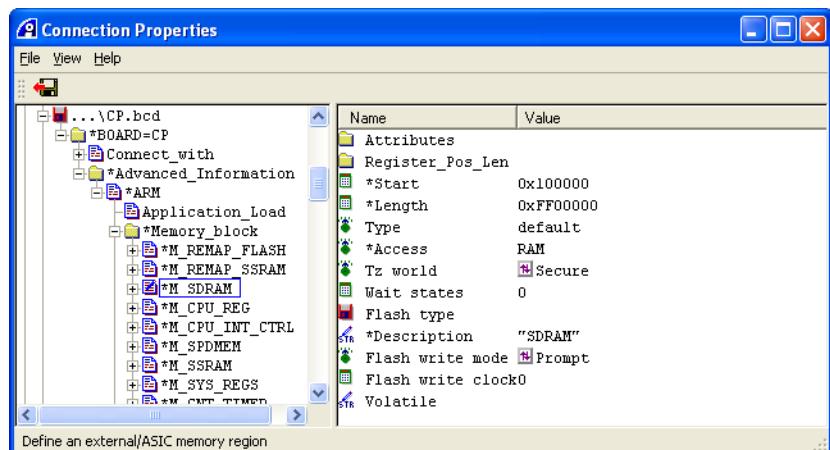


Figure 7-20 Editing the memory block in the CP Board/Chip definition file

To configure RealView Debugger for the target shown in Figure 7-19, you must set up several memory blocks. Each processor has a memory block for its private area and a block for a shared communication area. The ARM920T processor has two shared areas, so it has three memory blocks. You must create a separate group of settings for each processor.

Procedure for creating processor-specific connection settings

To create a separate Advanced_Information group for each processor:

1. Expand the Advanced_Information group for the Debug Configuration.
2. Create a new Advanced_Information block, and remove the Default block:
 - a. Right-click on Advanced_Information in the left pane to display the context menu.
 - b. Select **Make New...** from the context menu to display a rename dialog box.
 - c. Enter **ARM966EJ-S_2**.
 - d. Click **Create**.
 - e. Right-click on the Default block in the left pane.
 - f. Select **Delete**.

————— **Note** —————

RealView Debugger matches this name with the connection name when you connect to the processor. Therefore, only the Advanced_Information group settings that match the processor are assigned to that connection.

3. Make a copy of the ARM966EJ-S_2 group and name it **ARM920T_0**.
 4. Make a copy of the ARM920T_0 group and name it **ARM940T_1**.
- You now have a separate Advanced_Information block for each of the processors on your development platform.
5. Expand each processor group in turn, and set up the Memory_block with the following settings:

ARM966EJ-S_2 group

Create two sub-blocks named LocalMem and GfxMem in the Memory_block group. Change the settings in the groups as follows:

LocalMem	Set the following: Start=0 Length=0x10000 Description="Local program memory"
GfxMem	Set the following: Start=0x10000 Length=0x10000 Description="Frame Buffer" Set the following in the Attributes group: Shared=direct Shared_id=1

ARM920T_0 group

Create three sub-blocks named LocalMem, GfxMem, and CommsMem in the Memory_block group. Change the settings in the groups as follows:

LocalMem	Set the following: Start=0 Length=0x40000 Description="Local program memory"
GfxMem	Set the following: Start=0x40000 Length=0x10000 Description="Frame Buffer" Set the following in the Attributes group:

	Shared=direct Shared_id=1
CommsMem	Set the following: Start=0x80000 Length=0x1000 Description="Shared IPC memory"
	Set the following in the Attributes group: Shared=direct Shared_id=2

ARM940T_1

Create two sub-blocks named **LocalMem** and **CommsMem** in the **Memory_block** group.
Change the settings in the groups as follows:

LocalMem	Set the following: Start=0 Length=0x40000 Description="Local program memory"
CommsMem	Set the following: Start=0x80000 Length=0x1000 Description="Shared IPC memory" Set the following in the Attributes group: Shared=direct Shared_id=2

- Select **Save and Close** from the **File** menu to save your changes and close the Connection Properties window.

Observations on the three processor example

In the three processor example, the memory map for each processor is defined using the target name (for example, ARM920T) followed by an underscore and the TAP controller ID for that processor (for example, _0). Including the TAP number in addition to the processor name enables you to specify the exact processor if your development platform using more than one processor of the same type.

Within each processor memory block, some common properties of processor memory are defined, for example, defining the bus width using **Access_size**. These properties are inherited by the other memory specification blocks.

Specific properties, including start address and length of the memory regions, are defined for each of the memory regions. The regions are called **LocalMem**, **CommsMem** and **GfxMem**. In this example, the **CommsMem** region appears at the same place in the memory map of each of the processors accessing it, but the **GfxMem** does not. Where a shared region appears, in a given processor memory map, it is a function of the hardware memory address decoders on the target. It does not matter to RealView Debugger whether the shared regions map to the same addresses or to different addresses on the processors sharing them.

The default memory sharing state is not shared (indicated by the entry **Shared=none**), so the **LocalMem** definition does not have to state this. However, the **CommsMem** and **GfxMem** regions are shared, so the two attributes **Shared** and **Shared_id** must be specified for both regions. The value of **Shared** is one of:

- none** The memory region is not shared.
- direct** The memory region is shared.

The memory region share IDs used in this example are:

- 1 the video buffer memory
- 2 the interprocessor communications memory.

However, you can use any integer value in the range 0 to 65535.

Note

Although there is normally only one shared memory device, shared resources are described as part of the processor memory map, and not by physical device. RealView Debugger requires that the shared memory device is described at least twice, once for each processor sharing it. If you describe the device for only one processor, then RealView Debugger is unable to automatically update the views of the shared memory region for each processor.

Using Debug Interface units

The multiprocessor configurations described in this section can be implemented using a single DSTREAM or RealView ICE Debug Interface unit if all processors are on the same JTAG chain. When connected to multiple processors, the connection properties inherent in the interface apply to all the connections.

The configurations described can be also achieved using multiple DSTREAM or RealView ICE units to the same development platform.

Note

Although multiple target connections can be established using RVISS and ISSM models, the simulated targets cannot share memory, even if you configure a shared memory region. However, multiprocessor RTSM and SoC Designer models might contain memory that can be shared.

See also

- *Debug Interface units and multiple targets* on page 7-3
- the following in the *RealView Debugger Target Configuration Guide*:
 - *About configuring custom memory maps, registers, and peripherals* on page 4-2 for details on naming an Advanced_Information block.
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

7.11.6 Shared program memory

If you have program memory that is shared, then you might want to prevent software breakpoints being set in that region. To do this, set the **No_sw_bkpts** to **True** in the **Attributes** group for each definition of the shared memory block.

Chapter 8

Executing Images

There are several ways to control program execution from the RealView® Debugger Code window. These are described in the following sections:

- *About image execution* on page 8-2
- *Starting and stopping image execution* on page 8-4
- *Running an image to a specific point* on page 8-7
- *Stepping by lines of source code* on page 8-12
- *Stepping by instructions* on page 8-16
- *Stepping until a user-specified condition is met* on page 8-20
- *Resetting your target processor* on page 8-22.

8.1 About image execution

RealView Debugger enables you to control execution of your image. You can:

- start and stop execution
- step the image.

If your image displays messages or requires user input during execution, then RealView Debugger enables these messages and prompts to appear in the **StdIO** tab of the Output view.

Also, you can enter CLI commands directly or perform GUI operations that generate CLI commands when the image is running. However, if a CLI command cannot be actioned because the image is running, it is added to a command queue.

See also:

- *The stepping controls*
- *Interacting with your application* on page 8-3
- *The command queue* on page 8-3

8.1.1 The stepping controls

Controls are available for you to:

- step by lines of source code, referred to as high-level stepping
- step by assembly instructions, referred to as low-level stepping
- step until a user-specified condition is reached.

Stepping controls are provided:

- On the **Debug** menu, select:
 - **Step Into** (F11)
 - **Step Over (next)** (F10).
 The operations performed by these controls depend on the current view:
 - in the source code view, these controls perform high-level stepping
 - in the disassembly view, these controls perform low-level stepping.
- On the Code window Debug toolbar. There is a separate button for each type of step operation you can perform:
 - **Step** to step by line or instruction into functions
 - **Next** to step by line or instruction over functions.
- Using the various STEP commands. With the STEP commands, you can also:
 - step a specified number of source lines
 - step a specified number of assembly instructions.

See also

- *Controls for stepping by lines of code* on page 8-12
- *Stepping a specified number of source lines* on page 8-14
- *Controls for stepping by assembly instructions* on page 8-16
- *Stepping a specified number of assembly instructions* on page 8-18
- *Stepping until a user-specified condition is met* on page 8-20
- *Chapter 7 Debugging Multiprocessor Applications*.

8.1.2 Interacting with your application

During a debugging session, you can interact directly with your application using a mechanism called *semihosting*. Semihosting enables an application running on an ARM® architecture-based target to use the input and output facilities on a host computer that is running RealView Debugger. It is implemented by a set of defined *SuperVisor Call* (SVC) operations.

Standard semihosting is where the target processor enters debug state while the semihosting operation is performed, and is enabled by default for all ARM targets.

The behavior of semihosting depends on the target.

See also

- Standard semihosting behavior* on page 13-34.

8.1.3 The command queue

If an application is currently executing, RealView Debugger uses a command queue to handle any commands that cannot be executed immediately. Through this mechanism, commands build up on the queue and are then executed when resources become available. Commands are never executed out of order.

When an application is running and you request another action, then the corresponding command is added to the queue, pending execution. A message is displayed in the Output view to explain what has happened to the new command, for example:

```
> bexec (I A)0x000082CC
Command pended until execution stops. Use 'Cancel' to purge.
```

Be aware of the following:

- When a command has been queued, then any other command you enter, except for CANCEL, is added to the queue behind the pending command.
- Some commands are not supported when the image is executing (such as clearing a breakpoint). These commands are not added to the command queue, and a warning message is displayed. For example, if you attempt to clear a breakpoint, the following message is displayed:
Warning: This target does not support handling breakpoints while running.
- Any command that cannot be executed immediately, and that is allowed for a running target, is added to the queue.
- Breakpoints are set where possible, otherwise these commands are added to the queue. Typically, execution breakpoints are added to the command queue, but data access breakpoints are set.
- RealView Debugger appends unknown commands, and possibly invalid commands, to the command queue. No checking is done on the command syntax until the command is ready to be executed.
- Known invalid commands, for example those that do not start with a letter, are not added to the queue.

To clear, or purge, the command queue:

- Enter the CANCEL command.
- Select **Cancel Current Command** from the **Debug** menu.

8.2 Starting and stopping image execution

You must run an image to be able to monitor how it is executed on a target. Also, some debugging features require that you stop image execution.

See also:

- *Starting execution*
- *Stopping execution*
- *Considerations when running applications on RTSMs* on page 8-5.

8.2.1 Starting execution



To execute an image:

1. Connect to a suitable target.
2. Load an image to the target.
3. Click **Run** on the Debug toolbar.

This starts execution from the current location of the PC. The image runs until the program:

- ends
- encounters an error condition
- reaches a breakpoint
- prompts for user input
- stops because of a user action.

When execution has stopped, click **Run** again to resume program execution.

Running images linked using BE32 on RVISS ARM11 processor models

Images linked using the --BE32 option must be run on the _BE32 variant of any RVISS ARM11 processor model. All other images must be run on the other ARM11 models.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4.

8.2.2 Stopping execution



To stop image execution, click **Stop** on the Debug toolbar.

Be aware of the following if a semihosting operation is in progress:

- For RVISS:
 - You must not stop execution during semihosting input.
 - The stop operation is immediately actioned during semihosting output. You can restart execution again, if required.
- For ISSM:
 - the stop operation pends until you respond to the semihosting input
 - the stop operation is immediately actioned during semihosting output.

In both cases, you can restart execution again, if required.

- For RTSM:
 - the stop operation pends until you respond to the semihosting input
 - the stop operation is immediately actioned during semihosting output.

In both cases, you can restart execution again, if required.
- For a hardware target, the stop operation is immediately actioned. You can restart execution again, if required.

See also

- *Interacting with your application* on page 8-3.

8.2.3 Considerations when running applications on RTSMs

Be aware of the following when running applications on RTSMs.

The RTSM Rate Limit feature

The ARM RTSMs provided with *RealView Development Suite* (RVDS) Professional edition are highly optimized, and your code might run faster than on equivalent hardware. Therefore, a Rate Limit feature is supported so that you can enable or disable fast simulation. If Rate Limit is enabled, simulation time more closely matches real time.

To enable the Rate Limit feature, click the dark red square button in the top right of the Real-Time System Model CLCD window. The button changes to a bright red color. An example Real-Time System Model CLCD window is shown in Figure 8-1:



Figure 8-1 Real-Time System Model CLCD window

Running applications with graphical output on RTSMs

If your application outputs graphical information, the bottom half of the Real-Time System Model CLCD window expands to enable the graphical information to be displayed. Figure 8-2 on page 8-6 shows an example:

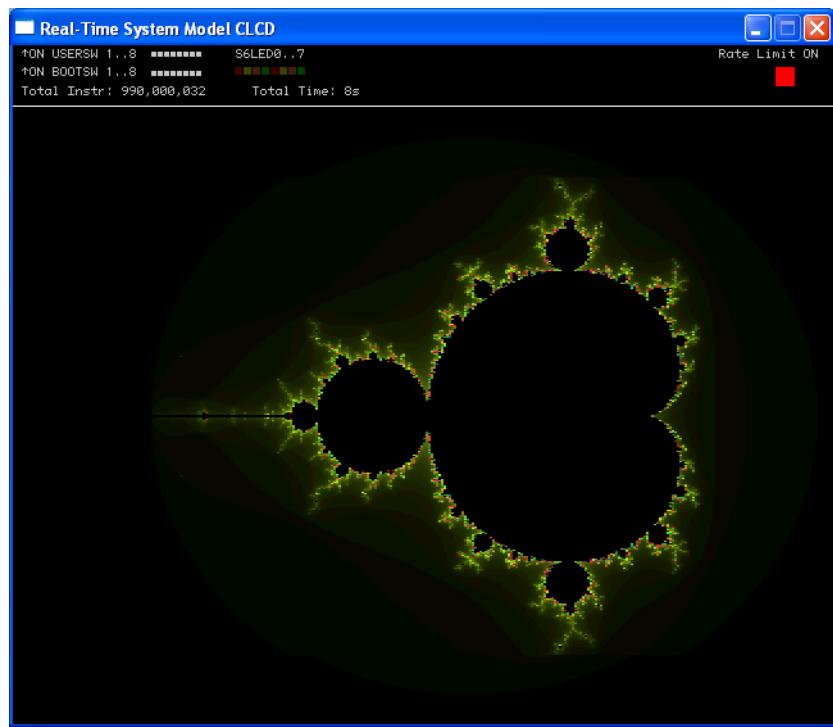


Figure 8-2 Real-Time System Model CLCD window with graphical output

Switching between execution and performance statistics

By default, the Real-Time System Model CLCD window shows the execution statistics. This appears on the third line of the display, for example:

Total Instr: 214,558,854 Total Time: 2s

Click on the third line of the display to switch to the performance statistics, for example:

Instr / sec: 76,549,873 Perf Index: 0.58

Note

Performance statistics are reset to zero when you stop execution.

See also

- *Connections to RTSMs* on page 3-29
- *RealView Development Suite Real-Time System Model User Guide*.

8.3 Running an image to a specific point

RealView Debugger gives you some control over how far an image runs before it stops, without having to set breakpoints.

Note

Be aware that when running to a specific point, RealView Debugger uses a temporary breakpoint to stop at the chosen point. However, if the chosen point is on ROM or Flash, RealView Debugger attempts to set a hardware breakpoint. If there are not enough breakpoint resources available, then RealView Debugger implements single instruction stepping, and displays the following warning:

Warning: Unable to set Software Break or Hardware Break at address, falling back to instruction stepping.

See also:

- *Running to the current cursor position*
- *Running to the start of a function using the source view* on page 8-8
- *Running to the start of a function using the Symbols view* on page 8-9
- *Running until the current function returns* on page 8-10
- *Running to an entry in the Call Stack* on page 8-11
- *Attaching a macro to the GO command* on page 16-20 for details on how to use a macro to control execution.

8.3.1 Running to the current cursor position

To run the image until execution reaches the code that corresponds to the current cursor position:

1. Load your application image. The rest of this procedure uses the example dhystone.axf image as an example.
2. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (dhry_1.c in this example).

Note

You can also use these controls in the disassembly view.

3. Locate the line of code where you want to stop execution. In this example, locate line 149 in dhry_1.c.
4. Click anywhere on the line of code to position the cursor at that line.

Note

In the disassembly view, locate the instruction of interest, and click anywhere on the line for that instruction.

5. Select **Run to Cursor** from the **Debug** menu. Execution starts from the current location of the PC and runs until it reaches a temporary breakpoint, defined by the cursor position (line 149 in this example).

8.3.2 Running to the start of a function using the source view

To run the image to the start of a function using the source view:

1. Locate a call to the required function in your source code.
2. Right-click on the function name, to display the context menu. For example, at line 149 of dhry_1.c, right-click on Proc_5.
3. Select **Locate Line...** from the context menu to display the Prompt dialog box containing the function name Proc_5.
4. Click **Set**. The dialog box closes and the start of the function (Proc_5 in this case) is displayed. A blue outlined arrow also shows the located line, shown in Figure 8-3.

———— Note ————

If the function is defined in a different source file, then RealView Debugger opens the source file.

```

File | dhry_1.c Find Line Scripts <None>
Home Page Disassembly dhry_1.c
374 Proc_5 () /* without parameters */
375 *****/
376 /* executed once */
377 {
378     Ch_1_Glob = 'A';
379     Bool_Glob = false;
380 } /* Proc_5 */
381
382
383     /* Procedure for the assignment of structures, */
384     /* if the C compiler doesn't support this feature */
385 #ifdef NOSTRUCTASSIGN
386 memcpy (d, s, 1)

```

Figure 8-3 Located line in the source code

5. Right-click in the margin next to the line indicated by the arrow, to display the context menu.
6. Select **Run To Here** from the context menu to start execution.

Where appropriate, enter the required input to any prompts displayed by the image. Your image runs until the start address of the chosen function is reached.

If the function is not defined in your source code, then there might be more than one level of call stack between the function and the code for which source is available. Therefore, the source that contains the nearest calling function is displayed, and execution appears to stop at the line of source for that function. You might have to click the **Disassembly** tab to see the address where the function is defined.

For example, if you choose the call to malloc at line 92 of dhry_1.c, then execution appears to stop at line 91. Click the **Disassembly** tab to see the address of the function code.

8.3.3 Running to the start of a function using the Symbols view

To run the image to the start of a function using the Symbols view:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 8-4 shows an example. If you have loaded multiple images to the target, then all the images are listed.

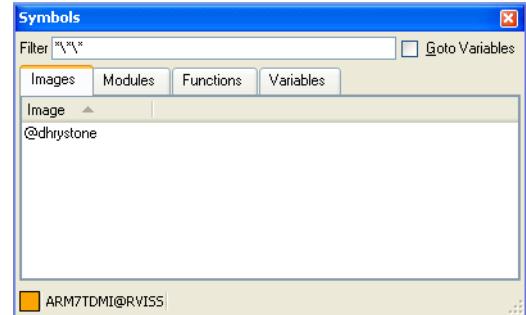


Figure 8-4 Symbols view

2. Click the **Functions** tab. This lists the public and static functions in the image, shown in Figure 8-5.

Function Name	Address	Scope	Module	Image
clock	0x000091AC	Public	SYSAPP	@dhystone
exit	0x0000BE8C	Public	STDLIB	@dhystone
fflush	0x0000C020	Public	STDIO	@dhystone
fgetc	0x0000CB90	Public	STDIO	@dhystone
fopen	0x0000BD74	Public	STDIO	@dhystone
fputc	0x0000CCB8	Public	STDIO	@dhystone
free	0x0000CB8A	Public	STDIO	@dhystone

Figure 8-5 List of functions

3. Locate the function where you want to stop execution.
4. Right-click on the function entry to display the context menu.
5. Select **Run To Here** from the context menu to start execution.

Where appropriate, enter the required input to any prompts displayed by the image. Your image runs until the start address of the chosen function is reached. When the function is reached, one of the following actions occurs:

- If the source file that defines the function is not open, then RealView Debugger opens the source file to show where execution has stopped.
- If the function is not defined in your source code, then execution appears to stop at the source line where the function is first called in your source code. You might have to click the **Disassembly** tab to see the address where the function is defined.

For example, if you choose the call to `malloc` at line 92 of `dhry_1.c`, then execution appears to stop at line 91. Click the **Disassembly** tab to see the address of the function code.

8.3.4 Running until the current function returns

To run the image until the current function returns:

1. Load your application image. The rest of this procedure continues to use the dhystone.axf image as an example.

You want to stop execution in the function of interest. To control the point at which the image stops, either:

- Locate the function in your source code, and use the **Run to cursor** control to run the image to a line of code in that function.
- Set a breakpoint in the function and then run the image in the usual way. This is the method used in the rest of this procedure.

2. Display line 149.
3. Right-click in the margin to display the context menu.
4. Set a breakpoint in the Proc_5() function:
 - a. Select **Create Breakpoint...** from the context menu to display the Create Breakpoint dialog box.
 - b. Enter **Proc_5** in the Location field.
 - c. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box.
-  5. Click **Run**. The program begins execution.
6. When prompted for the number of runs, enter **1000**. The program continues execution and runs up to the breakpoint. A red box shows the location of the PC at line 378.
The **Cmd** tab of the Output view shows where execution has stopped, for example:
Stopped at 0x000081E4 due to SW Instruction Breakpoint
Stopped at 0x000081E4: DHRY_1\Proc_5 Line 378
-  7. Click the **Run until return from current function** button on the Debug toolbar. Execution starts from the current PC in the current function (Proc_5 in this example) and runs until it returns.
For the Dhystone example, execution stops at the next line of code following the call to the Proc_5 function (line 150 of dhry_1.c).

See also

- *Starting and stopping image execution* on page 8-4
- *Running to the current cursor position* on page 8-7
- *Setting a breakpoint at the destination of a branch instruction* on page 11-37
- *Setting breakpoints on functions* on page 11-39.

8.3.5 Running to an entry in the Call Stack

To run the image to the point identified in a Call Stack entry:

1. Load your application image. The rest of this procedure continues to use the `dhystone.axf` image as an example.
2. Force the image execution to stop inside a function. For the Dhystone example:
 - a. Locate line 149 in the file `dhry_1.c`.
 - b. Select **Create Breakpoint...** from the context menu to display the Create Breakpoint dialog box.
 - c. Enter **Proc_5** in the Location field.
 - d. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box.
3. Run the image. For the Dhystone example, enter the value **5000** when prompted for the number of runs, the restart execution.
4. When execution stops, select **Call Stack** from the **View** menu, to display the Call Stack view, if it is not already visible. Figure 8-6 shows the Call Stack view for the Dhystone image.

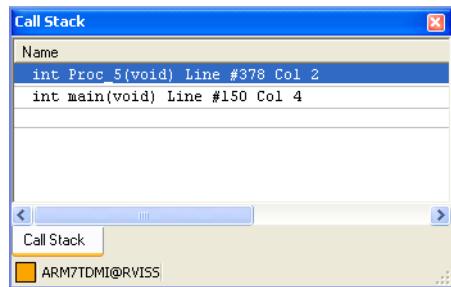


Figure 8-6 Example Call Stack view for dhystone

5. Right-click on the call stack entry for `main` to display the context menu.
6. Select **Run To** from the context menu. Execution continues until code associated with the call stack entry is reached. For the Dhystone image, execution stops at line 150 in `dhry_1.c`.

8.4 Stepping by lines of source code

There are times when you want to observe the sequential execution of your program. Stepping provides the fine control that enables you to do this.

Note

Be aware that when stepping at the source level, RealView Debugger uses temporary breakpoints to stop execution. However, when stepping in ROM or Flash, RealView Debugger attempts to set hardware breakpoints. If there are not enough breakpoint resources available, then RealView Debugger implements single instruction stepping, and displays the following warning:

Warning: Unable to set Software Break or Hardware Break at *address*, falling back to instruction stepping.

You must run the image until it stops before the area of interest. To control the point at which the image stops, either:

- run the image to a specific point
- set a breakpoint and then run the image in the usual way.

See also:

- *Controls for stepping by lines of code*
- *Stepping into functions* on page 8-13
- *Stepping over function calls* on page 8-13
- *Stepping a specified number of source lines* on page 8-14
- *Considerations when high-level stepping in the disassembly view* on page 8-14
- *Starting and stopping image execution* on page 8-4
- *Running an image to a specific point* on page 8-7
- *Chapter 11 Setting Breakpoints*.
- *Using a macro when stepping* on page 16-20 for details on how to use a macro when stepping.

8.4.1 Controls for stepping by lines of code

The following controls enable you to step by one line of code when you are viewing source code:

Step by line or instruction over functions

If the source line makes a call to a function, RealView Debugger executes the function completely before stopping, unless a breakpoint is met in the function call. Execution then depends on any breakpoint conditions and actions.

Step by line or instruction into functions

If the source line makes a function call, RealView Debugger steps into the function, unless there is no source code available for this function.

Note

If you perform a high-level step in code for which there is no source available, RealView Debugger attempts to step up the call stack until a location is reached that has source available.

See also

- *Breakpoints in RealView Debugger* on page 1-36.

8.4.2 Stepping into functions

To enter called functions when stepping:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
-  3. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (dhry_1.c in this example).
4. Display line 149.
5. Double-click on the line number in the left margin to set a breakpoint.

The following message is displayed in the Output view:

```
binstr \DHRY_1\#149:2
```

————— **Note** —————

The number following the colon : might be different to that shown. It is the position within the line where the breakpoint is set. This is significant only when setting a breakpoint on multistatement lines.



6. Click **Run**.

The program begins execution.

7. When prompted for the number of runs, enter **1000**.

The program continues execution and runs up to the breakpoint. A red box shows the location of the PC at line 149.

The **Cmd** tab of the Output view shows where execution has stopped, for example:

```
Stopped at 0x00008480 due to SW Instruction Breakpoint
Stopped at 0x00008480: DHRY_1\main Line 149
```



8. Click the **Step by line or instruction into functions** button once.

A red box shows the location of the PC at line 378. The Proc_5 function has been stepped into.

9. Click **Step by line or instruction into functions** several times, until you are familiar with the operation of this control.

The red box moves to the next line of code each time you click the button, even when the line of source comprises several instructions.

See also

- *Considerations when high-level stepping in the disassembly view* on page 8-14.

8.4.3 Stepping over function calls

To step without entering called functions:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
-  3. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (dhry_1.c in this example).
4. Display line 149.

5. Click on the Proc_5 function name.
6. Select **Run to Cursor** on the **Debug** menu. The program begins execution.
7. When prompted for the number of runs, enter **1000**. The program continues execution and runs to the position of the cursor in the source code. A red box shows the location of the PC at line 149.
-  8. Click the **Step by line or instruction over functions** button once. A red box shows the location of the PC at line 150. The Proc_5 function has been stepped over.
9. Click **Step by line or instruction over functions** several times, until you are familiar with the operation of this control.

The red box moves to the next line of code each time you click the button, without entering any called functions, even when the line of source comprises several instructions.

See also

- *Considerations when high-level stepping in the disassembly view.*

8.4.4 Stepping a specified number of source lines

To step a specified number of source lines, you must use the CLI commands shown in Table 8-1. Each command accepts an integer value that specifies the number of lines of code to execute during the multiple step operation. During stepping, the number of steps remaining is displayed in the Code window status bar, for example, 87 Step Remaining.

Table 8-1 Commands for stepping by lines of code

Step control	Code View	Command
Step by line or instruction into functions	Source	STEPLINE <i>n</i> (or STEP <i>n</i>)
Step by line or instruction over functions	Source	STEP0 <i>n</i>

See also

- *Considerations when high-level stepping in the disassembly view*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

8.4.5 Considerations when high-level stepping in the disassembly view

If you are performing high-level stepping in the disassembly view, then be aware of the following:

- RealView Debugger steps to the first instruction corresponding to each line of interleaved source code that is prefixed with **>>>**. Where a line of source code is implemented with multiple instructions, the remaining instructions are stepped over. Also, lines of interleaved source code prefixed with **---** are stepped over.

To step to lines of interleaved source prefixed with **---**, then enter the following command:

OPTION STEPPING=ALL

- If a breakpoint exists at an address spanned by the line of source, then that breakpoint might be activated if the breakpoint conditions are met during the step.
- If you are performing a high-level step over operation, any call to a function that is in your source code is stepped over.
- If you are performing a high-level step into operation, and a function is called that is in your source code, then that function is stepped into.
- The run state in the Code window status bar indicates *Waiting* when a stopped processor is in the middle of a synchronized multiple step operation, and it is waiting for another processor to stop before it continues with the multiple step.

See also

- *Breakpoints in RealView Debugger* on page 1-36
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the OPTION command.

8.5 Stepping by instructions

There are times when you want to observe the sequential execution of your program at the assembly instruction level. Stepping provides the fine control that enables you to do this.

Note

Be aware that when stepping instructions RealView Debugger uses a temporary breakpoint to stop execution. However, when stepping more than a single instruction in ROM or Flash, RealView Debugger attempts to set a temporary hardware breakpoint. If there are not enough breakpoint resources available, then RealView Debugger implements single instruction stepping, and displays the following warning:

Warning: Unable to set Software Break or Hardware Break at address, falling back to instruction stepping.

You must run the image until it stops before the area of interest. To control the point at which the image stops, either:

- run the image to a specific point
- set a breakpoint and then run the image in the usual way.

See also:

- *Starting and stopping image execution* on page 8-4
- *Running an image to a specific point* on page 8-7
- *Controls for stepping by assembly instructions*
- *Stepping into functions* on page 8-17
- *Low-level stepping over function calls* on page 8-18
- *Stepping a specified number of assembly instructions* on page 8-18
- *Considerations when low-level stepping in the source view* on page 8-19
- *Chapter 11 Setting Breakpoints*
- *Using a macro when stepping* on page 16-20 for details on how to use a macro when stepping.

8.5.1 Controls for stepping by assembly instructions

The following controls enable you to step by a single assembly instruction when you are viewing disassembly in the **Disassembly** tab:

Step by line or instruction into functions

If the assembly instruction makes a function call, RealView Debugger steps into the function.

Step by line or instruction over functions

If the assembly instruction makes a call to a function, RealView Debugger executes the function completely before stopping, unless a breakpoint is met in the function call. Execution then depends on any breakpoint conditions and actions.

See also

- *Breakpoints in RealView Debugger* on page 1-36.

8.5.2 Stepping into functions

To enter called functions when stepping assembly instructions:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
3. Locate the instruction in the **Disassembly** tab where you want to start stepping. You can either:
 - Scroll through the disassembly view in the **Disassembly** tab to locate the instruction.
 - Find an instruction corresponding to a line of code in a source file (dhry_1.c in this example).
4. Display line 149.
5. Set a breakpoint by double-clicking on the line number in the left margin. The following message is displayed in the Output view:

```
binstr \DHRY_1\#149:5
```

————— **Note** —————

The number following the colon : might be different to that shown. It is the position within the line where the breakpoint is set. This is significant only when setting a breakpoint on multistatement lines.



6. Click **Run**. The program begins execution.
7. When prompted for the number of runs, enter **1000**. The program continues execution and runs up to the breakpoint. A red box shows the location of the PC at line 149.
The **Cmd** tab of the Output view shows where execution has stopped, for example:
Stopped at 0x00008480 due to SW Instruction Breakpoint
Stopped at 0x00008480: DHRY_1\main Line 149
8. Right-click in the left margin of the source view at line 149 to display the context menu.
9. Select **Locate PC in Disassembly** from the context menu.

The code view changes to the **Disassembly** tab, and the assembly instruction where the breakpoint is set is displayed. A red box shows the location of the PC (address 0x8480 in this example).



10. Click the **Step by line or instruction into functions** button once.
A red box shows the location of the PC at address 0x81E4. The Proc_5 function has been stepped into.
11. Click **Step by line or instruction into functions** several more times, until you are familiar with the operation of this control.

See also

- *Considerations when low-level stepping in the source view* on page 8-19.

8.5.3 Low-level stepping over function calls

To step assembly instructions without entering called functions:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
3. Locate the instruction in the **Disassembly** tab where you want to start stepping. You can either:
 - Scroll through the disassembly view in the **Disassembly** tab to locate the instruction.
 - Find an instruction corresponding to a line of code in a source file (dhry_1.c in this example).
4. Display line 149.
5. Click on the Proc_5 function name.
6. Select **Run to Cursor** on the **Debug** menu. The program begins execution.
7. When prompted for the number of runs, enter **1000**. The program continues execution and runs to the position of the cursor in the source code. A red box shows the location of the PC at line 149.
8. Click the **Disassembly** tab. The code view changes to the **Disassembly** tab, and the assembly instruction where the breakpoint is set is displayed. A red box shows the location of the PC (address **0x8480** in this example).
-  9. Click the **Step by line or instruction over functions** button once. A red box shows the location of the PC at address **0x8484**. The Proc_5 function has been stepped over.
10. Click **Step by line or instruction over functions** several more times, until you are familiar with the operation of this control.

See also

- *Considerations when low-level stepping in the source view* on page 8-19.

8.5.4 Stepping a specified number of assembly instructions

To step a specified number of assembly instructions, you must use the CLI commands shown in Table 8-2. Each command accepts an integer value that specifies the number of instructions to execute during the multiple step operation. During stepping, the number of steps remaining is displayed in the Code window status bar, for example, 87 Step Remaining.

Table 8-2 Commands for stepping by assembly instructions

Step control	Code view	Command
Step by line or instruction into functions	Disassembly	STEPI <i>n</i>
Step by line or instruction over functions	Disassembly	STEPOI <i>n</i>

See also

- *Considerations when low-level stepping in the source view* on page 8-19

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

8.5.5 Considerations when low-level stepping in the source view

If you are performing low-level stepping in the source view, then be aware of the following:

- Where lines of source code comprise several assembly instructions, the red box does not move from the source line until all instructions for that line of source have been executed.
- If a breakpoint is hit when stepping, then the breakpoint might be activated if the breakpoint conditions are met during the step operation.
- The run state in the Code window status bar indicates *Waiting* when a stopped processor is in the middle of a synchronized multiple step operation, and it is waiting for another processor to stop before it continues with the multiple step.

See also

- *Breakpoints in RealView Debugger* on page 1-36.

8.6 Stepping until a user-specified condition is met

There are times when you want to see what happens when various parts of an image get executed. You can step through the code by specifying a condition, in the form of a C-style expression. When the condition is met, that is the condition is nonzero, execution halts.

Note

Be aware of the following:

- Using this feature causes execution to slow down and might result in errors because of timing issues.
- If a breakpoint is hit during a step operation, then the breakpoint might stop execution before the step operation completes, depending on the breakpoint conditions.
- When stepping RealView Debugger uses temporary breakpoints to stop execution. However, when stepping in ROM or Flash, RealView Debugger attempts to set temporary hardware breakpoints. If there are not enough breakpoint resources available, then RealView Debugger implements single instruction stepping, and displays the following warning:

Warning: Unable to set Software Break or Hardware Break at address, falling back to instruction stepping.

You must run the image until it stops before the area of interest. To control the point at which the image stops, either:

- run the image to a specific point
- set a breakpoint and then run the image in the usual way.

To step until a specific condition is met:

1. Connect to your target.
2. Load the required image, for example `dhrystone.axf`.
3.  Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (`dhrystone_1.c` in this example).
4. Display line 149.
5. Click on the `Proc_5` function name.
6. Select **Run to Cursor** on the **Debug** menu. The program begins execution.
7. When prompted for the number of runs, enter **1000**. The program continues execution and runs to the position of the cursor in the source code. A red box shows the location of the PC at line 149.
8. Select **Step Until Condition...** from the **Debug** menu to display the Prompt dialog box.
9. Enter an expression to test, using the standard C expression format. Enter **Run_Index==10** for this example.
10. Click **OK**. The dialog box closes, and execution stops when `Run_Index` equals 10.

Note

Step Until Condition... steps into called functions. Therefore, execution might stop at a line of code within a called function.

See also:

- *Breakpoints in RealView Debugger* on page 1-36
- *Running an image to a specific point* on page 8-7
- *Starting and stopping image execution* on page 8-4
- Chapter 11 *Setting Breakpoints*.

8.7 Resetting your target processor

Where supported by your debug target, you might want to reset your target processor during a debugging session. The reset might be hard or soft depending on the processor type. See your processor hardware documentation for details.

To reset a processor:

1. Select **Process Control** from the **View** menu to display the Process Control view.
2. Right-click on the top-level process entry to display the **Process** context menu.
3. Select **Reset Target** to perform the reset.

See also:

- *Considerations when resetting a processor.*

8.7.1 Considerations when resetting a processor

Be aware that the status of the target after a reset depends on the Debug Interface configuration. For example, target connections through a DSTREAM or RealView ICE unit might be affected if you have changed the Reset parameters in the Advanced panel of the RVConfig utility.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 3 *Customizing a Debug Configuration*
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

Chapter 9

Mapping Target Memory

This chapter describes how to map target memory in RealView® Debugger. It includes:

- *About mapping target memory* on page 9-2
- *Setting the default access type for unmapped memory regions* on page 9-5
- *Enabling memory mapping* on page 9-6
- *Viewing the memory map* on page 9-8
- *Setting up a temporary memory map* on page 9-12
- *Setting up a memory map* on page 9-15
- *Creating a temporary memory map entry* on page 9-18
- *Editing a memory map entry* on page 9-20
- *Updating the memory map* on page 9-22
- *Deleting memory map blocks* on page 9-23
- *Generating linker command files for non-ARM targets* on page 9-24.

9.1 About mapping target memory

Mapping target memory is the mechanism that RealView Debugger provides to specify the memory map of a development platform. You can define a memory region as:

- ROM
- RAM
- Flash
- *Write-Only Memory* (WOM)
- *No Memory* (NOM).

Alternatively, a memory region can be defined as Auto memory, where RealView Debugger decides what type or memory to assign to a region when you load an image. Therefore, through RealView Debugger, you can have full access to all the available memory on your debug target. You can define the memory map of your development platform:

- in one or more BCD files, which you assign to a Debug Configuration
- in the **Memory Map** tab of the Process Control view
- at the RealView Debugger CLI by:
 - manually entering commands
 - loading a script file.

See also:

- *Uses for memory mapping*
- *Memory mapping with RVISS models* on page 9-3
- *Considerations when using memory maps* on page 9-3

9.1.1 Uses for memory mapping

Mapping target memory is an important step of the debug process, and provides many benefits to debugging your application. The primary benefits of mapping target memory are as follows:

- You can define all target memory types. That is, RAM, ROM, WOM, and Flash memory, including the access size.
- The debugger can correctly load an image to your target hardware. Flash memory is programmed with information you must define in the BCD and FME files.
- The debugger can inform you when you try to load an image into memory that does not exist on your target hardware.
- You can define shared memory, which is updated by the debugger when modified by a different processor.
- You can generate a linker command file for targets that are not ARM® architecture-based.

See also

- *Sharing resources between multiple targets* on page 7-28
- *Setting up a memory map* on page 9-15
- *Generating linker command files for non-ARM targets* on page 9-24
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger*.

9.1.2 Memory mapping with RVISS models

Although you can assign BCD files to a *RealView ARMulator® Instruction Set Simulator* (RVISS) configuration, they do not enable access to the RVISS map file related statistics and wait states. To access these statistics and wait states, you must set up an RVISS map file.

When you connect to an RVISS model, the RVISS memory map is displayed in the **Mapfile** tab of the Registers view.

You can also access the map file related statistics and wait states using the symbol `@mapfile_symbolname`, where `symbolname` is the name of the symbol. A Mapfile name symbol has any non-alphanumeric characters converted to underscores.

Note

If you want to create a map file that is unique to a specific RVISS Debug Configuration, then you must create a variant processor model.

See also

- *RVISS map file related statistics* on page 13-75
- *Viewing the map related statistics on the Mapfile tab* on page 13-81
- *RealView ARMulator ISS User Guide* for details on how to:
 - set up an RVISS map file
 - create a variant processor model.

9.1.3 Considerations when using memory maps

When working with memory mapping, you must be aware of the following:

- Make sure you do not set `top_of_memory` outside the regions that you have defined as valid in your memory map.
- The default ARM C library implementation of the function `__user_setup_stackheap()` makes a semihosting call to read the debugger variable `top_of_memory`. The value returned is used to locate the stack base of your application. RealView Debugger uses target-dependent defaults for the stack and heap limits. For more details about the `__user_setup_stackheap()` function, see the following documentation:
 - *ARM® Compiler toolchain Using ARM® C and C++ Libraries and Floating-Point Support*
 - *ARM® Compiler toolchain ARM® C and C++ Libraries and Floating-Point Support Reference*
 - *ARM® Compiler toolchain Using the Linker*
 - *ARM® Compiler toolchain Linker Reference*.

Note

The `__user_setup_stackheap()` function replaces the `__user_initial_stackheap()` function in the ARM C library.

Also, see the trace project provided with *RealView Development Suite* (RVDS) for an example of how to use this function.

- When an image is loaded to a debug target, RealView Debugger checks the memory map to confirm that it is valid to load to the locations specified in the executable program. Memory is loaded and then read back to verify a successful load and to confirm that genuine memory is present. Memory sections defined as Auto are also updated to reflect the access type specified in the executable image.
- When you set a breakpoint in RealView Debugger, by default RealView Debugger tries to set a software breakpoint. However, your memory map determines the type of breakpoint that can be set in memory:
 - hardware breakpoints can be set in Flash, RAM, ROM, and WOM
 - software breakpoints can only be set in RAM.
- The different types of memory (that is, Flash, RAM, ROM, and WOM) are color coded when displayed in the Memory view.

Considerations for targets that support TrustZone technology

For targets that support TrustZone® technology:

- The memory map shows both the Secure World and the Normal World memory maps.
- Specify the TrustZone world for a memory region (the default is Secure). How you do this depends on the method you use to create the map entry:
 - When using the Create Map Entry and Edit Map Entry dialog boxes to set up temporary memory map entries, prefix the address with either S: or N:.
 - When setting up memory map entries in a BCD file, set the Tz_world setting to the appropriate value and specify addresses in the memory block without the S: or N: prefix.

9.2 Setting the default access type for unmapped memory regions

Unmapped memory regions are identified as Default Mapping in the **Memory Map** tab of the Process Control view. By default, all such memory regions have an access type of No Memory (NOM). However, you can change the default so that all unmapped memory regions have an access type of RAM.

— Note —

You must disconnect and reconnect the target for the setting to take effect.

To change the default access type of all unmapped memory regions:

1. Disconnect from the target if a connection is established.
2. Select **File** → **Workspace** → **Workspace Options...** from the Code window main menu. The Workspace Options window is displayed.
3. Expand the **DEBUGGER** group in the left pane.
4. Select the **Memory_maps** group in the left pane.
5. In the right pane, change Access to the required value, either:
 - **NOM** (the default)
 - **RAM**.
6. Select **Save and Close** from the **File** menu.
7. Reconnect to the target.

See also

- *Disconnecting from a target* on page 3-52
- *Configuring workspace settings* on page 17-15
- *Memory_maps* on page A-4.

9.3 Enabling memory mapping

Memory mapping is disabled by default when you first connect to an ARM architecture-based debug target. In this case, RealView Debugger treats all memory as RAM, which might be undesirable for your application. To allow RealView Debugger to correctly debug your target, you must enable memory mapping and specify the memory regions on your target.

To enable memory mapping:

1. Before you can view a memory map, connect to a target (see *Connecting to a target* on page 3-27).
2. Select **Memory Map Tab** from the **View** menu to display the Process Control view and bring the **Memory Map** tab to the front. Figure 9-1 shows an example:

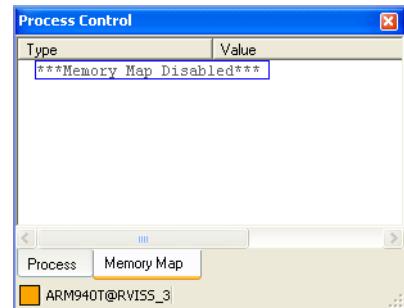


Figure 9-1 Disabled memory map

Alternatively, if you have an instance of the Process Control view that is docked, click the **Memory Map** tab in that view.

3. Right-click anywhere in the **Memory Map** tab to display the context menu.
4. Select **Memory Mapping** from the context menu.

A check mark is displayed to the left of the option, to indicate that memory mapping is enabled. Also, a default memory map entry is displayed, as shown in Figure 9-2.

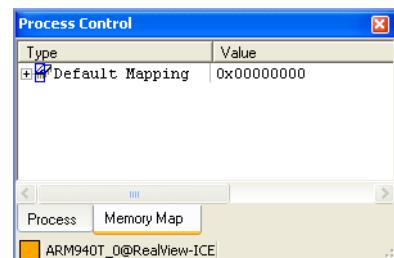


Figure 9-2 Default memory map

See also:

- *Disabling memory mapping* on page 9-7
- *Considerations when using memory maps* on page 9-7
- *Updating the memory map* on page 9-22.

9.3.1 Disabling memory mapping

To disable memory mapping:

1. Right-click anywhere in the **Memory Map** tab to display the context menu.

———— Note ————

The context menu contains different options depending on the state of your debug target. For example, if an image is loaded the menu contains the option **Update Map Based on Image**.

2. Select **Memory Mapping** from the context menu.

The check mark to the left of the option is removed, to show that memory mapping is disabled, and the memory map is hidden.

9.3.2 Considerations when using memory maps

Be aware of the following when memory maps are enabled:

- The memory map is updated when you load an image.
- A memory map can be configured in a *board/chip definition* (BCD) file that you associate with the connection. If you connect to a target that is configured in this way, then memory mapping is automatically enabled.
- Any areas of memory that are not defined by the memory map are shown as default mapping. If target memory exists in these areas, it is treated as RAM. If any of these memory areas are not RAM (for example, ROM or Flash), then this can affect various operations you perform in RealView Debugger. For example, if you attempt to set a software breakpoint in ROM, then an error occurs. However, you can force the creation of a hardware breakpoint using the failover qualifier, depending on the remaining hardware breakpoint resources available.
- If you delete any or all memory map blocks in the **Memory Map** tab that have been defined in a BCD file, then that memory map is no longer available for the connection. However, the definitions in the BCD file are not deleted. Therefore, to restore the memory map back to that defined in the BCD file, you must disconnect and then reconnect.

See also

- *How loading an image affects the memory map* on page 9-11
- *Setting up a memory map* on page 9-15
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *BREAKINSTRUCTION* on page 2-55.

9.4 Viewing the memory map

The Process Control view provides a view of the memory mapping for the debug target that is running your application.

To view the memory map:

1. Connect to a target.
2. Enable memory mapping.
3. Select **Memory Map Tab** from the **View** menu to display the Process Control view and bring the **Memory Map** tab to the front. Figure 9-3 shows an example:

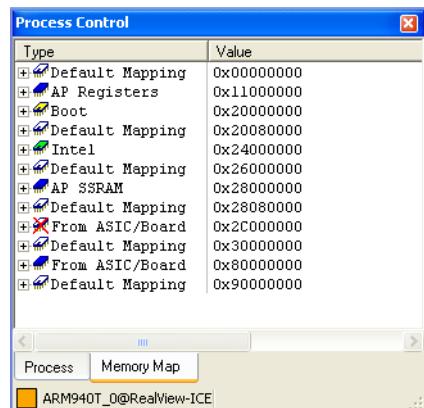


Figure 9-3 Example of a memory map

Alternatively, if you have an instance of the Process Control view that is visible, click the **Memory Map** tab in that view.

The **Memory Map** tab displays a tree-like structure for each region of the memory map showing the description and start address. Additional details for each region include the size and access rule, as shown in see Figure 9-5 on page 9-10). The way that memory is shown depends on your debug target because RealView Debugger populates this tab from:

- built-in knowledge about the target
- any BCD files assigned to the Debug Configuration
- loaded images.

Colored icons are used to show the type of memory defined.

With an image loaded, the **Memory Map** tab is updated from details in the image itself. The memory map is also automatically updated if any registers change that affect memory mapping.

See also:

- *Memory maps on targets that support TrustZone technology* on page 9-9
- *Memory map entry* on page 9-9
- *Display colors* on page 9-10
- *How loading an image affects the memory map* on page 9-11
- *Connecting to a target* on page 3-27
- *Enabling memory mapping* on page 9-6
- *Display colors* on page 9-10
- *How loading an image affects the memory map* on page 9-11
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Setting up controlled memory blocks* on page 4-49.

9.4.1 Memory maps on targets that support TrustZone technology

For a target that supports TrustZone technology, the memory map shows map entries for both the Secure World and the Normal World. By default, all memory map entries defined in BCD files are defined for the Secure World. Figure 9-4 shows an example:

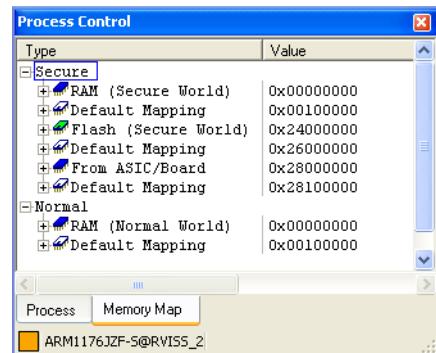


Figure 9-4 Memory map for a target that supports TrustZone technology

9.4.2 Memory map entry

An entry in the **Memory Map** tab contains the following:

Entry name This is the name of the map entry as defined by:

- the Description setting in the BCD file Memory_block definition
- the Description field of the Create Map Entry and the Edit Map Entry dialog boxes.

Entry address

The start address of the map entry.

Size The size of the map entry.

Access The access rule for the map entry:

RAM Memory is readable and writable.

ROM Memory is read-only.

WOM Memory is write-only.

NOM No memory.

Auto Memory is defined by the application currently loaded. If there is no application loaded, this shows NOM.

Prompt You are prompted to confirm that this type of memory is permitted for the loaded application. If there is no application loaded, this shows NOM.

Flash Flash memory is readable and, if a Flash programming method file (*.fme) is present, writable.

To program Flash in RealView Debugger, memory mapping must be enabled, and a Flash memory map entry must be present that references the correct *Flash MMethod* (FME) file.

Filled Details of those parts of an image that has been loaded into the map entry. If no image is loaded into a map entry, this group is not shown. The group lists the following details:

- the name of the loaded image

- each image section loaded into the entry, together with the address range that each section occupies.

To see details about a map entry, right-click on the chosen entry and select **Display Properties** from the context menu. This displays a text description of the type of memory defined at this location.

See also

- *How loading an image affects the memory map* on page 9-11
- *Creating a temporary memory map entry* on page 9-18
- *Editing a memory map entry* on page 9-20
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*
 - *Memory mapping Advanced_Information settings reference* on page A-20.

9.4.3 Display colors

When using the **Memory Map** tab to view the memory map, RealView Debugger uses colored icons to highlight the different memory regions. An example memory map is shown in Figure 9-5.

The screenshot shows the 'Process Control' window of the RealView Debugger. The main pane displays a hierarchical tree of memory mappings under the 'Type' column. The 'Value' column shows the starting address for each mapping. Colored icons next to the entries indicate the memory access type:

Type	Value
+ Default Mapping	0x00000000
AP Registers	0x11000000
Size	0x0F000000
Access	RAM
Boot	0x20000000
Size	0x00080000
Access	ROM
+ Default Mapping	0x20080000
Intel	0x24000000
Size	0x02000000
Access	Flash
+ Default Mapping	0x26000000
AP SSRAM	0x28000000
Size	0x00080000
Access	RAM
+ Default Mapping	0x28080000
From ASIC/Board	0x2C000000
Size	0x04000000
Access	NOM
+ Default Mapping	0x30000000

At the bottom of the window, there are tabs for 'Process' and 'Memory Map'. The 'Memory Map' tab is selected, indicated by a yellow bar. Below the tabs, the text 'ARM940T_0@RealView-ICE' is displayed.

Figure 9-5 Colors in the memory map

Colored icons enable you to identify the memory access defined:

- white (open) indicates one of the following:
 - no memory is defined, that is, RealView Debugger has assigned this by default
 - you have set the memory access to Auto, that is, auto-define as RAM when you load an image into it
 - you have set the memory access to Prompt, that is, prompt if you load an image into it.
- blue indicates RAM
- yellow indicates ROM
- green indicates Flash

- red indicates write-only memory (WOM)
- red cross indicates no accessible memory (NOM) is defined.

9.4.4 How loading an image affects the memory map

When you load an image and memory mapping is enabled, the memory map is updated with the properties of the image, such as:

- the memory region occupied by the image, and the size of the region (a scatterloaded image might occupy more than one memory region)
- the image sections, and the memory region occupied by each section.

Figure 9-6 shows the memory map when the dhystone.axf example image is loaded.

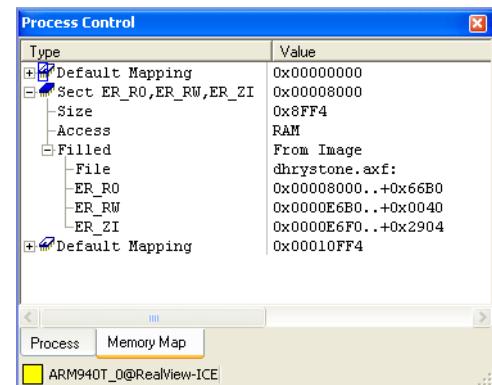


Figure 9-6 Memory map with an image loaded

9.5 Setting up a temporary memory map

You can set up a memory map for your current debug session. Target memory settings defined in this way are only temporary and are lost when you disconnect from the target.

To set up a temporary map:

1. Create a Debug Configuration that does not have a BCD file assigned to it.
2. Connect to a target in the Debug Configuration.
3. Enable memory mapping.
4. Right-click in the **Memory Map** tab, shown in Figure 9-7, to display the context menu.

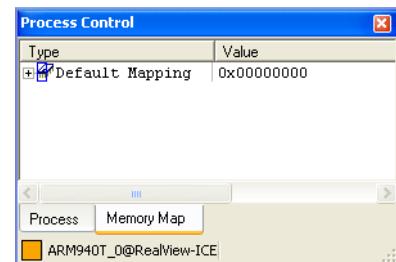


Figure 9-7 Default memory map

5. Select **Create Map Entry...** from the context menu to display the Create Map Entry dialog box. Figure 9-8 shows an example:

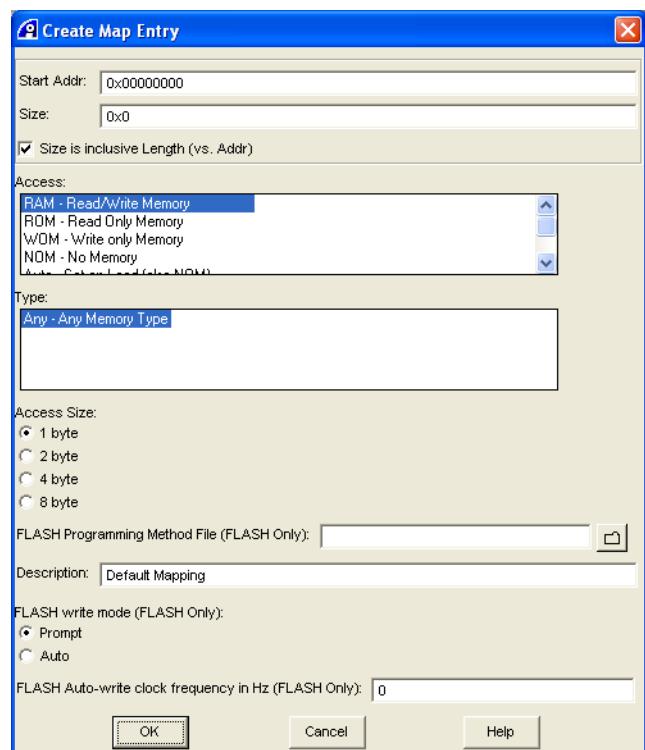


Figure 9-8 Create Map Entry dialog box

Note

In RealView Debugger, memory mapping is defined by a start address and a block size by default, not by an end address. If you want to specify the end address, you must deselect the **Size is inclusive Length (vs Addr.)** check box.

6. Specify the starting location for the new map entry in the Start Addr field.
For this example, leave the address as **0x0**.
7. Specify the block size for the new map entry in the Size field.
For this example, enter **0x8000**.
By default, this specifies the size of the memory block to be defined. To specify the absolute end address, rather than the size:
 - a. Deselect the **Size is inclusive Length (vs. Addr)** check box.
 - b. Enter the required address in the Size field. For example, enter **0xFFFFFFFF0**.
RealView Debugger automatically sets the size you specify. If the computed size does not fall on a page boundary an error dialog is displayed and you must re-enter the block size.
If you enter a value of **0x0**, all memory from the starting address is remapped.
8. Select the access type in the Access list.
For this example, leave the type as RAM – Read/Write Memory.
9. Select the memory type to be allocated from the Type list. The options listed depend on your target.
10. If required, select the size of memory accesses in bytes from the Access Size.

Note

By default, memory access is set to byte-size (8 bits) for a Default Mapping entry in the **Memory Map** tab.

11. If you selected Flash – FLASH/EEPROM Updateable as the Access type, then you must specify the appropriate Flash MEthod (FME) file to use for your development platform.
Otherwise, skip this step.

To specify the FME file to use for Flash programming:

- a. Click the file browser button for the FLASH Programming Method File (FLASH Only) field.
- b. Select <Select File ...> from the menu to display the Select File dialog box.
- c. Locate the required FME file.
- d. Click **Open**.

The FME file and path are inserted into the FLASH Programming Method File (FLASH Only) field.

You can optionally set the FLASH write mode to one of the following:

Prompt Causes the Flash Memory Control dialog box to be displayed so that you can manually perform a Flash write operation.

Auto Causes RealView Debugger to perform Flash write operations automatically.
Also, set the FLASH Auto-write clock frequency in Hz field to the correct speed.

In this mode, the write operation is complete when RealView Debugger displays the Flash Memory Control dialog box. Also, messages are displayed in the **Cmd** tab of the Output view, depending on the feature used to perform the write operation.

12. Enter a description for this entry in the Description field.
For this example, enter **Area before image**.
13. Click **OK** to confirm your changes. The memory map is updated to show the memory block you have added.
14. Repeat these steps for other memory map entries you want to define.
For this example, create two map entries with the settings shown in Table 9-1.

Table 9-1 Settings for new map entries

Setting	Second Block	Third block
Start address	0x8000	0x10000
Length	0x8000	0xFFFF0000
Description	Middle	Area after image

———— Note ————

If you right-click on the Default Mapping entry, and select **Create Map Entry...**, then the start address for the new entry is set to the next unused address.

15. Expand each memory map entry to see the details. The memory map looks like that shown in Figure 9-9.

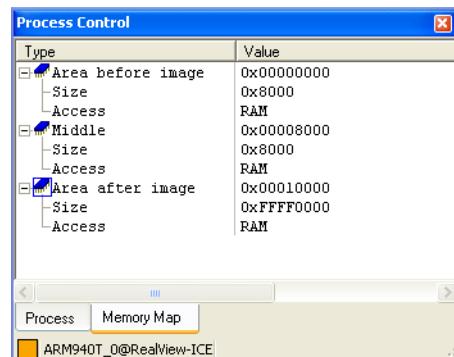


Figure 9-9 Example of a memory map

See also:

- *About creating a Debug Configuration* on page 3-8
- *Connecting to a target* on page 3-27
- *Enabling memory mapping* on page 9-6.
- *Chapter 6 Writing Binaries to Flash*
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Chapter 4 Configuring Custom Memory Maps, Registers and Peripherals* for details of how to configure a memory map as part of your Debug Configuration.
 - *Chapter 6 Programming Flash with RealView Debugger*.

9.6 Setting up a memory map

You can set up a custom memory map and memory mapped registers by using a BCD file containing board/chip definitions. You assign the board/chip definitions to a Debug Configuration before you connect.

BCD files for some common components, such as ARM development boards and ARM core modules, are supplied with RealView Debugger. If any of the supplied BCD files correspond to components of your development platform, then you can use those BCD files. Otherwise, you must create your own BCD file.

See also:

- *Assigning board/chip definitions to a Debug Configuration*
- *Views that are affected by assigning BCD files* on page 9-16
- *Considerations when using memory maps* on page 9-17
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

9.6.1 Assigning board/chip definitions to a Debug Configuration

To assign a board/chip definition to a Debug Configuration:

1. Disconnect all connections that are currently established on the specific Debug Configuration.
2. In the Connect to Target window, right-click on the Debug Configuration that you want to customize.
3. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.

———— Note ————

In the remaining steps of this procedure you are going to assign the CP and CM966ES board definitions to demonstrate how to assign board/chip definitions to a connection.

4. Click the **BCD files** tab to display the available definitions and assignments.
5. To assign board/chip definitions to this configuration:
 - a. Select the CP definition in the Available Definitions list.
 - b. Click the **Add** button.
The CP definition is moved from the Available Definitions list to the Assigned Definitions list.
 - c. Repeat these steps to assign the CM966ES definition.
6. Click **OK** to save your changes.
7. Connect to the target.

This has the effect described in *Views that are affected by assigning BCD files* on page 9-16.

9.6.2 Views that are affected by assigning BCD files

The memory map for this example is shown in Figure 9-10. Some memory map entries are defined in the CM966ES.bcd file, the remainder are defined in the CP.bcd file.

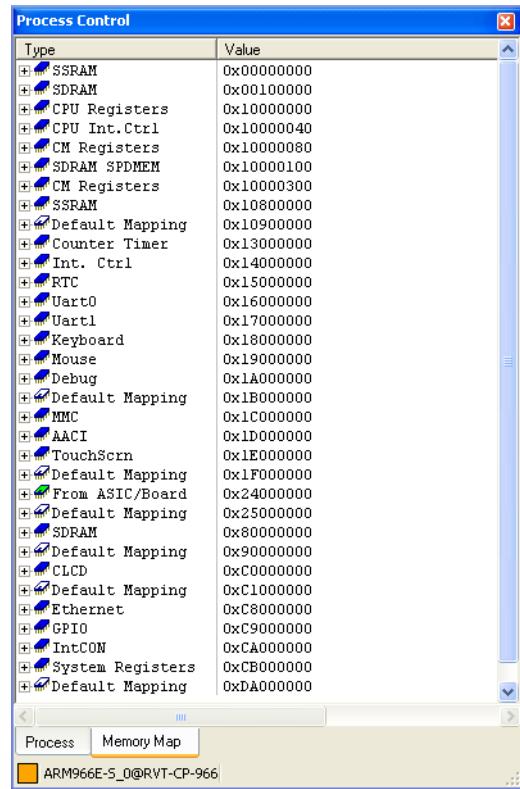


Figure 9-10 Integrator/CP and ARM966E-S memory map

When connected to the target, a **CP** tab and a **CM966ES** tab are provided in the Registers view, shown in Figure 9-11.

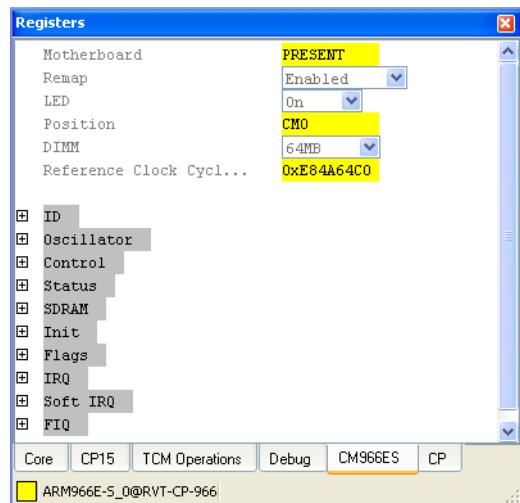


Figure 9-11 CP and CM966ES tabs in the Registers view

9.6.3 Considerations when using memory maps

Be aware of the following:

- Changing or deleting any entries for a memory map in the **Memory Map** tab does not affect the memory map defined in the BCD file. If you want to restore the memory map for a target, you must disconnect and then reconnect.
- If a BCD file that defines a memory map is assigned to a Debug Configuration, then memory mapping is automatically enabled when you connect to a target in that Debug Configuration.

See Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals* in the *RealView Debugger Target Configuration Guide* for details of configuring your target this way.

- If a BCD file is assigned to a Debug Configuration, but no tab is visible for the corresponding target, make sure you have assigned the correct BCD file for that target. For example, if a CM940T bcd file is assigned to the Debug Configuration, and you connect to an ARM966E-S processor, no tab is displayed. Change the BoardChip_name setting in the Connection Properties from CM940T to CM966ES. Alternatively, if the Debug Configuration has an ARM940T processor and an ARM966E-S processor, then add a new BoardChip_name setting with the value CM966ES.

9.7 Creating a temporary memory map entry

You might have a memory map defined for the connected target (for example, in a BCD file) that does not exactly match the memory map of that target. In this situation, you can create a temporary memory map entry in the existing memory map. This enables you to see the effects of the memory map addition before you modify the BCD file. The temporary changes are not applied to any assigned BCD files.

To create a temporary memory map entry:

1. Right-click on the required map entry in the **Memory Map** tab to display the context menu.
2. Select **Create Map Entry...** to display the Create Map Entry dialog box. Figure 9-8 on page 9-12 shows an example.

The current settings for the chosen map entry are inserted into the appropriate fields.

3. Specify the starting location for the new map entry in the Start Addr field.
4. Specify the block size for the new map entry in the Size field.

By default, this specifies the size of the memory block to be defined. To specify the end address, rather than the block size:

- a. Deselect the **Size is inclusive Length (vs. Addr)** check box.
- b. Enter the required address in the Size field. For example, enter **0xFFFFFFFF0**.

RealView Debugger automatically sets the size you specify. If the computed size does not fall on a page boundary an error dialog is displayed and you must re-enter the block size.

If you enter a value of **0x0**, all memory from the starting address is remapped.

5. Select the access type in the Access list.
For example, select **RAM - Read/Write Memory**.
6. Select the memory type to be allocated from the Type list. The options listed depend on your target.
7. If required, select the size of memory accesses in bytes from the Access Size.
8. If you selected **Flash - FLASH/EEPROM Updateable** as the Access type, then you must specify the appropriate *Flash MMethod* (FME) file to use for your development platform. Otherwise, skip this step.

To specify the FME file to use for Flash programming:

- a. Click the file browser button for the FLASH Programming Method File (FLASH Only) field.
- b. Select **<Select File ...>** from the menu to display the Select File dialog box.
- c. Locate the required FME file.
- d. Click **Open**.

The FME file and path are inserted into the FLASH Programming Method File (FLASH Only) field.

You can optionally set the FLASH write mode to one of the following:

Prompt Causes the Flash Memory Control dialog box to be displayed so that you can manually perform a Flash write operation.

Auto Causes RealView Debugger to perform Flash write operations automatically. Also, set the FLASH Auto-write clock frequency in Hz field to the correct speed.

In this mode, the write operation is complete when RealView Debugger displays the Flash Memory Control dialog box. Also, messages are displayed in the **Cmd** tab of the Output view, depending on the feature used to perform the write operation.

9. Enter a description for the new memory map block, for example `New test memory entry`.
10. Click **OK** to confirm your new settings and to update the **Memory Map** tab.

———— **Note** ————

If you have entered any values that are inconsistent, for example there is a mismatch on start and end addresses, RealView Debugger displays a warning. Correct these entries and click **OK**. When all entries are valid, the dialog box closes and RealView Debugger updates the **Memory Map** tab.

See also:

- Chapter 6 *Writing Binaries to Flash*
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 6 *Programming Flash with RealView Debugger*.

9.8 Editing a memory map entry

You can edit any memory map entry except for those that have default mapping. If the memory map is defined in a BCD file, then the changes are not applied to the related BCD file.

To edit a memory map entry:

1. Right-click on the required map entry in the **Memory Map** tab to display the context menu.
For example, if you have created the temporary memory map as described in *Setting up a temporary memory map* on page 9-12, right-click on the **Area after image** entry.
2. Select **Edit Map Entry...** to display the Edit Map Entry dialog box. Figure 9-12 shows an example. The current settings for the chosen map entry are inserted into the appropriate fields.

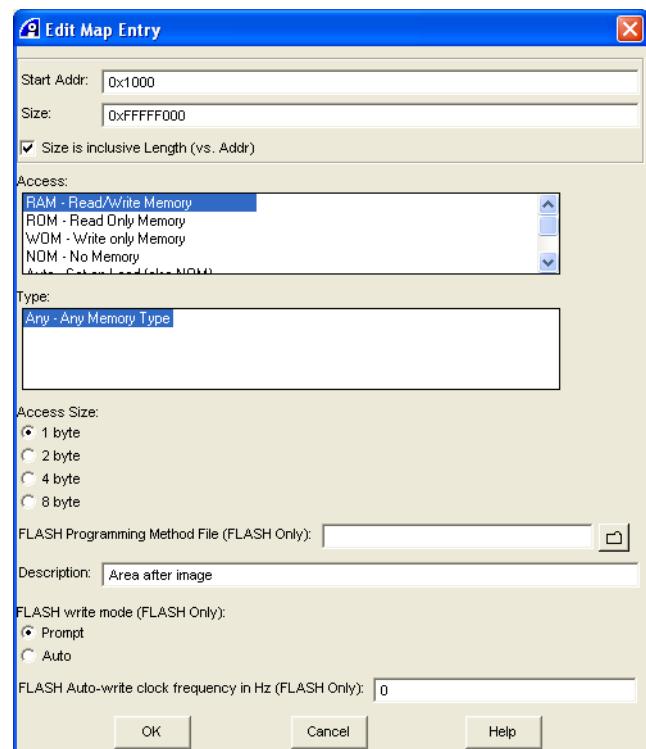


Figure 9-12 Edit Map Entry dialog box

3. Specify the starting location for the map entry in the Start Addr field.

4. Specify the size for the map entry in the Size field.

By default, this specifies the size of the memory block being edited. To specify the end address rather than the block size:

- a. Deselect the **Size is inclusive Length (vs. Addr)** check box.

- b. Enter the required address in the Size field. For example, enter **0xFFFFFFFF0**.

RealView Debugger automatically sets the size you specify. If the computed size does not fall on a page boundary an error dialog is displayed and you must re-enter the block size.

If you enter a value of **0x0**, all memory from the starting address is remapped.

5. Select the access type in the Access list.

For example, select **ROM - Read Only Memory**.

6. Select the memory type to be allocated from the Type list. The options listed depend on your target.
7. If you selected Flash - FLASH/EEPROM Updateable as the Access type, then you must specify the appropriate *Flash MMethod* (FME) file to use for your development platform. Otherwise, skip this step.

To specify the FME file to use for Flash programming:

- a. Click the file browser button for the FLASH Programming Method File (FLASH Only) field.
- b. Select <Select File ...> from the menu to display the Select File dialog box.
- c. Locate the required FME file.
- d. Click **Open**.

The FME file and path are inserted into the FLASH Programming Method File (FLASH Only) field.

You can optionally set the FLASH write mode to one of the following:

Prompt Causes the Flash Memory Control dialog box to be displayed so that you can manually perform a Flash write operation.

Auto Causes RealView Debugger to perform Flash write operations automatically. Also, set the FLASH Auto-write clock frequency in Hz field to the correct speed.

In this mode, the write operation is complete when RealView Debugger displays the Flash Memory Control dialog box. Also, messages are displayed in the **Cmd** tab of the Output view, depending on the feature used to perform the write operation.

8. Enter a description for the new memory map block, for example `New test memory entry`.
9. Click **OK** to confirm your new settings and to update the **Memory Map** tab.

————— **Note** —————

If you have entered any values that are inconsistent, for example a mismatch on start and end addresses, RealView Debugger displays a warning. Correct these entries and click **OK**. When all entries are valid, the dialog box closes and RealView Debugger updates the **Memory Map** tab.

See also:

- *Setting up a temporary memory map* on page 9-12
- *Chapter 6 Writing Binaries to Flash*
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Chapter 6 Programming Flash with RealView Debugger*.

9.9 Updating the memory map

The memory map is usually updated automatically by RealView Debugger when you:

- connect to a target that has a BCD file specified in the parent Debug Configuration
- load an image with memory mapping enabled.

However, you might have to manually update the memory map in some situations, for example, if you are using custom memory mapped registers.

To update the memory map:

1. Right-click in the **Memory Map** tab to display the context menu.
2. Select one of the following from the context menu:

Update Map Based on Image

To update the memory map based on details held in the loaded image.

This is done automatically when you load an image.

Update Map Based on Processor

To update the memory map based on those registers that affect memory maps.

This is done automatically for built-in map registers but might be required if you are using external map registers, defined in the Debug Configuration settings. Select this option to force RealView Debugger to read the registers and so update the memory map.

If you are connected to a debug target that uses register-controlled remapping, for example the ARM Integrator™/CP board, the **Memory Map** tab displays the effects of any changes made to these registers. In this case, use this option to update the display based on these memory-mapped registers.

See also:

- *How loading an image affects the memory map* on page 9-11
- *Setting up a memory map* on page 9-15.

9.10 Deleting memory map blocks

You can delete memory map blocks either individually or as a single operation:

- If you have defined a temporary memory, then all your changes are lost.
- If the memory map is defined in a BCD file that is assigned to the connection, then the deleted blocks apply for as long as the connection is established. If you disconnect then reconnect, the memory map defined in any assigned BCD files is restored.

See also:

- *Deleting all memory map blocks*
- *Deleting a specific memory map block.*

9.10.1 Deleting all memory map blocks

To delete all memory map blocks:

1. Right-click in the **Memory Map** tab to display the context menu.
2. Select **Delete All** from the context menu. All memory map entries are deleted, and replaced with the default entry. Figure 9-13 shows an example:

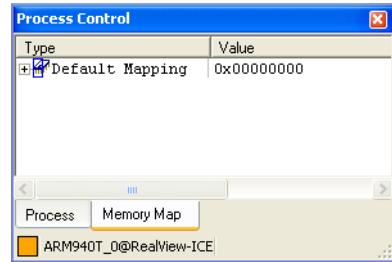


Figure 9-13 Default memory map

9.10.2 Deleting a specific memory map block

To delete a specific memory map block:

1. Right-click on the memory map block to be deleted to display the context menu.
2. Select **Delete Map Entry** from the context menu. The memory map block is deleted. If the delete memory block leaves a gap in the memory map, then RealView Debugger insert a Default Mapping entry for that area of memory.

9.11 Generating linker command files for non-ARM targets

The memory map, shown in the **Memory Map** tab of the Process Control view, can be used to generate or modify a MEMORY section of a linker command file used when you build your program. This MEMORY directive information can then be used to position various sections of an application correctly.

— Note —

Although you can save the memory map for an ARM target, the linker command file cannot be used when building images for ARM architecture-based processors.

To generate or modify a linker command file:

1. Right-click on the start address at the top of the entries and select **Save Map to Linker Command File...** from the context menu.
2. Specify the location of the file in the Select Linker Command File to Create or Modify dialog box. Remember that:
 - If the file already exists, RealView Debugger looks for a MEMORY directive block created previously and, if found, replaces that block.
 - If the file already exists, but no MEMORY directive block exists, RealView Debugger locates the first MEMORY section and inserts the MEMORY directive block before it.
 - If the file already exists, RealView Debugger makes a backup copy before updating the contents.
 - If there is no existing file, RealView Debugger creates the specified file ready to accept the MEMORY directive block.

The RealView Debugger linker command file generation process uses the built-in automatic memory mapping to generate data based on the connected target settings, for example the registers that control mapping.

The data recorded in the generated MEMORY block includes each internal RAM, ROM, and Flash section as appropriate. Each section is allocated a predefined name. All external memory added using the **Memory Map** tab, or defined automatically from a loaded image, is allocated a name based on the characteristics of the memory.

— Note —

If a file already exists and contains a RealView Debugger generated data block then this section is replaced when RealView Debugger updates the command file.

Chapter 10

Changing the Execution Context

This chapter describes how to change the execution context during a debugging session. It contains the following sections:

- *About changing the execution context* on page 10-2
- *Changing scope to the PC* on page 10-5
- *Displaying the current execution context* on page 10-6
- *Resetting the PC to the image entry point* on page 10-7
- *Setting the PC to the address of an instruction or line of code* on page 10-8
- *Setting the PC to a function* on page 10-9
- *Changing scope to the code pointed to by a Call Stack entry* on page 10-12.

10.1 About changing the execution context

RealView® Debugger enables you to define the current execution context, and to change the context if required.

See also:

- *What is scope and context?*
- *Why change scope?*
- *Methods of forcing scope* on page 10-3
- *Code views* on page 10-3.

10.1.1 What is scope and context?

RealView Debugger uses *scope* to determine the value of a symbol. Scope defines how RealView Debugger accesses variables and resolves symbols in expressions. The scope of your program determines the execution *context*. Any symbol value available to a C or C++ program at the current PC location is also available to RealView Debugger.

When your program is executing, the PC stores the address of the instruction that is being prefetched. By default, the scope is set when the PC changes. Loading an image sets the PC at the entry point using *autoscope*, that is the PC defines the scope. Autoscope is also used in an assembly language routine when you step into code that has no source information.

RealView Debugger uses a yellow arrow and red box to highlight the PC location in the selected code view. The PC is only visible in an execution tab, specifically a source file tab or the **Disassembly** tab. However, RealView Debugger enables you to *force* scope to a different location.

10.1.2 Why change scope?

During execution, your application stores function calls on the call stack. If execution has stopped in a called function (because of a breakpoint, for example), then you can perform a traceback through the call stack. That is, you can change scope to a previous function. This enables you to examine any variables that are within the scope of that function.

See also

- *Changing scope to the code pointed to by a Call Stack entry* on page 10-12.

10.1.3 Methods of forcing scope

Program scope is determined by the PC value. If you want to change the program scope, you can force scope by setting it to an entry on the Call Stack.

Note

Forcing scope does not change the PC.

When you force scope to a location, RealView Debugger:

- Places a filled blue arrow at that location.
 - Issues the CLI command SCOPE, and displays the following message in the Output view:

```
> scope context
Scoped to: (address): module\function
```
- You can also use the CLI command CONTEXT to display the current scope location.

Note

Stepping changes scope to the location that is stepped to.

See also

- *Changing scope to the PC* on page 10-5
- *Changing scope to the code pointed to by a Call Stack entry* on page 10-12
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the CONTEXT and SCOPE commands.

10.1.4 Code views

When RealView Debugger first loads an image, and assuming that you do not force scope, the Code window contains tabs showing program execution:

- A source file tab shows the current context, that is the location of the PC at the entry point. As the context moves to lines in other source files, RealView Debugger opens those files. Therefore, you can track program execution when the context is within your source code.
- The **Disassembly** tab displays disassembled code with intermixed C/C++ source lines and, if available, the location of the PC. You can track program execution in the disassembly view.

Note

If you are tracking program execution in the source view, and the context moves to code that does not have an associated source file, RealView Debugger automatically displays the disassembly view.

Figure 10-1 on page 10-4 shows an example:

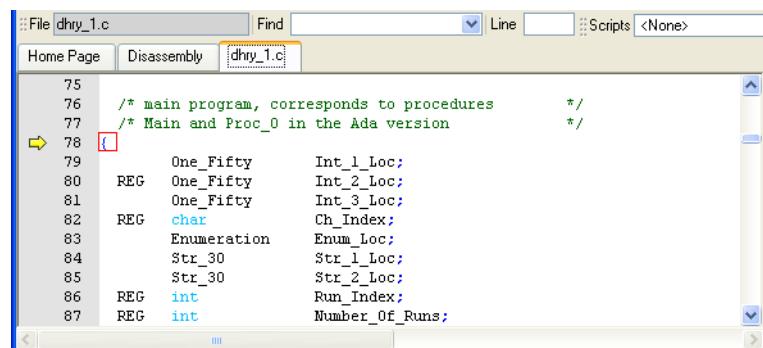


Figure 10-1 Code and disassembly tabs in the Code window

When you click the **Locate PC** button on the Debug toolbar, and source exists at the scope, RealView Debugger:

- opens the source file, if the source file is not currently open
- brings the statement where the PC is located into view
- draws a red box around that statement to highlight it in the code view
- displays a yellow arrow in the grey margin.

If no source exists at the current scope, RealView Debugger:

- changes the view to the **Disassembly** tab
- brings the corresponding line of disassembly into view
- draws a red box around that line to highlight it in the disassembly view
- displays a yellow arrow in the grey margin.

Note

Be aware that RealView Debugger also changes scope when you are examining captured trace and the Code Window Tracking feature is enabled.

See also

- the following in the *RealView Debugger Trace User Guide*:
 - *Viewing the captured trace* on page 9-11.

10.2 Changing scope to the PC

By default, the PC defines the current execution context. However, if you have forced scope to another location, you might want to restore the scope to the PC.

To change the scope to the PC, right-click on the background in the current code view to display the context menu, and select either:

- **Locate PC in Source**
- **Locate PC in Disassembly.**

Table 10-1 shows the command that is issued in each code view when you select these options from the context menu.

Table 10-1 Commands issued when locating PC in each code view

Option used	Disassembly View	Source view
Locate PC in Disassembly	SCOPE	MODE
Locate PC in Source	MODE	SCOPE

When RealView Debugger issues the MODE command, the code view changes to the alternate view.

When RealView Debugger issues the SCOPE command, the scope changes to the current PC location, and the following is displayed in the **Cmd** tab of the Output view:

At the PC: (address): module\function

However, if you are viewing a source file, and no source exists at the PC, the view changes to the **Disassembly** tab and the corresponding line of disassembly is displayed.

See also:

- *Methods of forcing scope* on page 10-3
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the MODE and SCOPE commands.

10.3 Displaying the current execution context

If you have been performing many operations, including forcing scope, then you might not remember where you last set the current execution context. RealView Debugger enables you to determine the current execution context.

To display the current execution context, enter the CONTEXT command. The current execution context is displayed in the Output view, for example:

```
> context  
Scoped to: (0x0000BF1C): STDIO\fflush
```

If the context is at the PC, and source exists at the PC, then the context also shows the location in your source, for example:

```
> context  
At the PC: (0x00008000): STARTUP_S\__mainSource view: DHRY_1\main Line 78
```

See also:

- *Methods of forcing scope* on page 10-3
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the CONTEXT command.

10.4 Resetting the PC to the image entry point

After executing an image, you can reset the PC to the image entry point without having to reload the image. This is useful when you do not want to reset the values of variables, reset the *Stack Pointer* (SP), or clear breakpoints. Scope is also changed to the PC.

Note

If there are any global or static variables that are initialized in the image, then they are not reset to their initial values.

To reset the PC to the image entry point, either:

- Select **Set PC to Entry Point** from the **Debug** menu.
- Click **Set PC** on the Debug toolbar.



This has the following effect:

- If the Registers view is displayed, the color of the PC register value changes to blue to indicate that the value has changed.
- The code view changes to show the code at the new PC location.
- A red box is drawn around the entry point instruction or line of code, to indicate the new PC location.
- The Call Stack view changes to show the new context, but does not have any traceback information.
- The **Locals**, **Statics**, and **This** tabs in the Locals view show the variables that are in scope at the new context.
- A RESTART command is issued.

See also:

- *Methods of forcing scope* on page 10-3
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the RESTART command.

10.5 Setting the PC to the address of an instruction or line of code

When viewing code in the **Disassembly** tab or a source file, you can set the PC to the address of an instruction or a line of code in your source file.

To set the PC to the address of an instruction or line of code:

1. Right-click in the gray margin at the instruction or line of code to display the context menu.
2. Select **Set PC to Here** from the context menu.

This has the following effects:

- If the Registers view is displayed, the color of the PC register value changes to blue to indicate that the value has changed.
- The PC changes to the address of the chosen instruction or first address of the line of code.
- A red box is drawn around the chosen instruction or line of code, to indicate the location of the PC.
- The Call Stack view changes to show the new context, but does not have any traceback information.
- The **Locals**, **Statics**, and **This** tabs in the Locals view show the variables that are in scope at the new context.
- A SETREG command is issued.

See also:

- *Locating the line of code using a symbol in the source view* on page 5-6
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the SETREG command.

10.6 Setting the PC to a function

You can set the PC to either the:

- Start address of a function.
- Entry point of a function, which identifies the first location in the function after the code that sets up the function parameters. In general, this is either:
 - the first executable line of code in that function
 - the first local variable that is initialized in that function.

Note

This does not change the current scope.

See also:

- Setting the PC to a function using the Symbols view*
- Setting the PC to a function entry point* on page 10-11.

10.6.1 Setting the PC to a function using the Symbols view

To set the PC to a function using the Symbols view:

- Connect to your target.
- Load an image, for example dhystone.axf.
- Select **Symbols** from the **View** menu to display the Symbols view. Figure 10-2 shows an example:

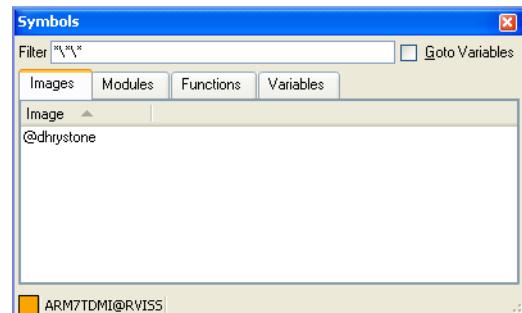


Figure 10-2 Symbols view

- Click the **Functions** tab to view the functions for the image. Figure 10-3 shows an example:

Symbols				
Filter <input type="text"/> Goto Variables				
Function Name	Address	Scope	Module	Image
clock	0x000091AC	Public	SYSAPP	@dhystone
exit	0x0000BE8C	Public	STDLIB	@dhystone
fflush	0x0000C020	Public	STDIO	@dhystone
fgetc	0x0000CB90	Public	STDIO	@dhystone
fopen	0x0000BD74	Public	STDIO	@dhystone
fputc	0x0000CB88	Public	STDIO	@dhystone
foo	0x00000000	Public	UCAPI	@dhystone

Figure 10-3 Functions in the dhystone.axf image

5. Locate the function where you want to set the PC.

_____ Note _____

You can use the Filter field to apply a filter on the complete list of functions. Enter the characters (in uppercase or lowercase) that are common to the required functions.

For example, to list all functions that begin with the letter P, then enter **\P or **\p.

6. Right-click on the function name to display the context menu. For example, right-click on the **fflush** function.
7. Select **Set PC to Here** from the context menu. The PC changes to the first location of the function (at address 0xBF14 in this example). Figure 10-4 shows an example:

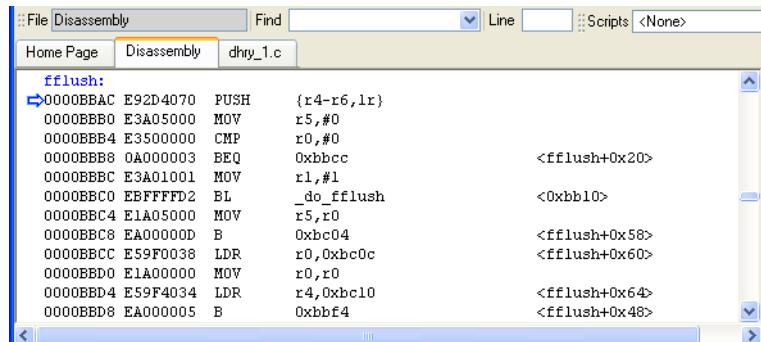


Figure 10-4 Set PC to a function

This has the following effect:

- If the Registers view is displayed, the color of the PC register value changes to blue.
- The code views change to show the code at the new PC location.
- A red box is drawn around the chosen instruction or line of code, to indicate the location of the PC.
- The Call Stack view changes to show the new context, but does not have any traceback information.
- The **Locals**, **Statics**, and **This** tabs in the Locals view show the variables that are in scope at the new context.
- A SETREG command is issued.

See also

- *Changing scope to the PC* on page 10-5
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the SETREG command.

10.6.2 Setting the PC to a function entry point

To set the PC to a function entry point, use the CLI command SETREG, and qualify the function name with the symbol @entry:

```
setreg @pc=function\@entry
```

This has the following effect:

- If the Registers view is displayed, the color of the PC register value changes to blue.
- The code views change to show the code at the new PC location.
- A red box is drawn around the chosen instruction or line of code, to indicate the location of the PC.
- The Call Stack view changes to show the new context, but does not have any traceback information.
- The **Locals**, **Statics**, and **This** tabs in the Locals view show the variables that are in scope at the new context.

For example, to set the PC to the entry point of the Proc_8() function in the dhystone.axf example image, enter the command:

```
setreg @pc=Proc_8\@entry
```

This sets the PC to the address 0x8FAC (line 97 in dhry_2.c).

Compare this with the following command:

```
setreg @pc=Proc_8
```

This sets the PC to the address 0x8FA4, which is the first address of the function (line 93 in dhry_2.c).

See also

- *Changing scope to the PC* on page 10-5
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Constructing expressions* on page 1-14 for details about the @entry symbol
 - *Alphabetical command reference* on page 2-12 for details of the SETREG command.

10.7 Changing scope to the code pointed to by a Call Stack entry

You can change scope to the location of a Call Stack entry whenever execution is halted.

Note

After changing scope to another Call Stack entry, you can examine the variables that are in scope using the **Locals**, **Statics**, and **This** tabs in the Locals view.

You must run the image until it stops either before a specified area of interest, or by some error. To control the point at which the image stops, either:

- run or step the image to a specific point
- set a breakpoint and then run the image in the usual way.

See also:

- *Changing scope to a specific entry in the Call Stack*
- *Moving up and down the Call Stack* on page 10-14.

10.7.1 Changing scope to a specific entry in the Call Stack

To change scope to a specific entry in the Call Stack:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
3. Click the **Locate PC** button on the Debug toolbar to view the source file that contains the PC scope (dhry_1.c in this example).
4. Locate the point in your code where you want to stop execution. In this example, set a breakpoint in the Proc_8() function, as follows:
 - a. Display line 167 in dhry_1.c.
 - b. Right-click on the Proc_8 function name to display the context menu.
 - c. Select **Create Breakpoint...** from the context menu to display the Create Breakpoint dialog box.
 - d. Enter **Proc_8** in the Location field.
 - e. Click **OK**.
5. Click **Run**. The program begins execution.
6. When prompted for the number of runs, enter **1000**. The program continues execution and runs up to the breakpoint. A yellow arrow and red box shows the location of the PC at line 93. Figure 10-5 shows an example:

```

File dhry_2.c Find Scripts <None>
Home Page Disassembly dhry_2.c dhry_1.c
90 Arr_2_Dim Arr_2_Par_Ref;
91 int Int_1_Par_Val;
92 int Int_2_Par_Val;
93 ( ) REG One_Fifty Int_Index;
94 REG One_Fifty Int_Loc;
95
96
97 Int_Loc = Int_1_Par_Val + 5;
98 Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
99 Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
100 Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
101 for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)

```

Figure 10-5 Stopped at breakpoint in Proc_8() function

The **Cmd** tab of the Output view shows where execution has stopped, for example:

Stopped at 0x00008FA4 due to SW Instruction Breakpoint

Stopped at 0x00008FA4: DHRY_1\main Line 93

7. Display the Call Stack view if it is not already visible. Figure 10-6 shows the Call Stack view for this example. The scope is located at line 93 in dhry_2.c.

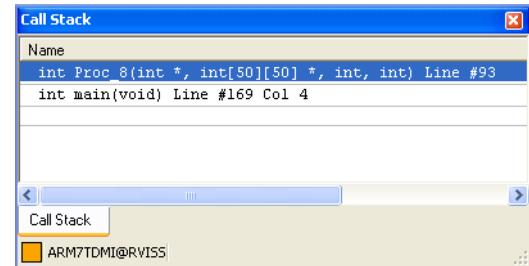


Figure 10-6 Call Stack view

8. Double-click on an entry in the Call Stack view. RealView Debugger:

- changes scope to the corresponding line of code.
- displays a message in the Output view showing the new context.

In this example, double-click on the `int main(void) Line #169` entry:

- The new scope is located at line 169 in dhry_1.c
- The following message is displayed in the **Cmd** tab of the Output view:
Scoped at level 1: (0x00008520): DHRY_1\main Line 169
- A blue arrow and blue box shows the location of the scope. Figure 10-7 shows an example:

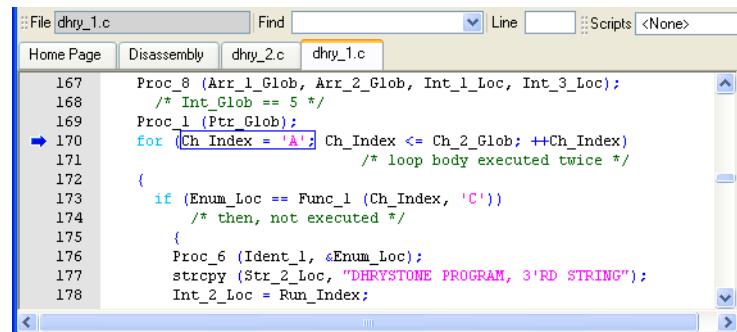


Figure 10-7 Scoped to a Call Stack entry

See also

- [Starting and stopping image execution](#) on page 8-4
- [Viewing variables for the current context](#) on page 13-14
- [Chapter 8 Executing Images](#)
- [Chapter 11 Setting Breakpoints](#).

10.7.2 Moving up and down the Call Stack

You can change scope by moving up and down the Call Stack. To do this, use the UP and DOWN CLI commands.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the UP and DOWN commands.

Chapter 11

Setting Breakpoints

This chapter explains the different types of breakpoints supported by RealView® Debugger, describes the options for setting breakpoints, and explains how to manage breakpoints during your debugging session. It includes:

- *About setting breakpoints* on page 11-3
- *Setting a simple breakpoint* on page 11-13
- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Clearing breakpoints* on page 11-18
- *Viewing breakpoint information* on page 11-20
- *Disabling a breakpoint* on page 11-22
- *Enabling a breakpoint* on page 11-23
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Finding a breakpoint in the code view* on page 11-30
- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints by dragging and dropping* on page 11-32
- *Setting breakpoints on lines of source code* on page 11-34
- *Setting breakpoints on instructions* on page 11-37
- *Setting breakpoints on functions* on page 11-39
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Forcing the size of a software breakpoint* on page 11-62
- *Chaining hardware breakpoints* on page 11-63

- *Specifying processor exceptions (global breakpoints)* on page 11-65
- *Setting breakpoints on custom memory mapped registers* on page 11-67
- *Setting breakpoints from the breakpoint history list* on page 11-70
- *Creating new breakpoint favorites* on page 11-72
- *Setting breakpoints from your Favorites List* on page 11-74.

11.1 About setting breakpoints

RealView Debugger enables you to set different types of breakpoints when you are debugging your image. You can set:

- software and hardware breakpoints
- breakpoints that are unconditional or conditional.
- breakpoints that are specific to OS-aware applications.

The type of breakpoint you can set depends on the:

- memory map, if enabled
- hardware support provided by your target processor
- Debug Interface used to maintain the target connection
- running state if you are debugging an OS-aware application.

Note

RealView Debugger also uses temporary breakpoints to halt execution as you:

- run to a specific point in your application
- step through your application.

These breakpoints are not visible to you, and RealView Debugger removes them when execution continues.

See also:

- *Software and hardware breakpoints*
- *Breakpoint types for OS-aware connections* on page 11-6
- *Unconditional and conditional breakpoints* on page 11-6
- *Breakpoints in different memory map regions* on page 11-8
- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Help with specifying locations and values for hardware breakpoints* on page 11-11
- *Breakpoints and image restarts* on page 11-11

11.1.1 Software and hardware breakpoints

RealView Debugger enables you to set software or hardware breakpoints, depending on your target memory type.

Software breakpoints

These breakpoints are placed by RealView Debugger by temporarily modifying program memory with a special instruction opcode. Because RealView Debugger requires write access to program memory, software breakpoints can only be set in RAM.

Note

In memory that is shared between multiple processors it is not advisable to use software breakpoints, because they are hit on both targets.

Hardware breakpoints

Hardware breakpoints are implemented by EmbeddedICE® logic that monitors the address and data buses of your processor. For simulated targets, hardware breakpoints are implemented by your simulator software. The complexity and number of breakpoints you can set depends on:

- hardware support provided by your target processor
- the Debug Interface used to maintain the target connection.

Note

RealView Debugger can reserve one breakpoint unit for internal use (for example, when one or more software breakpoints are set). Therefore, there might be one fewer hardware breakpoint available to you. An error message is displayed if you try to set a hardware breakpoint when the limit is reached.

If advanced breakpoint support is provided by your target or simulator software, for example *RealView ARMulator® ISS* (RVISS), you can set more complex hardware breakpoints. These breakpoints might be data-dependent or take advantage of range functionality. For example, some targets enable you to chain two breakpoints together, so that the breakpoint activates only when the conditions of both breakpoints are met.

Check your vendor-supplied documentation, to determine the breakpoint capability of your hardware, if supported.

Note

You cannot set breakpoints on core registers. However, you can set breakpoints on memory mapped registers.

The number of hardware breakpoints supported by your debug target is both architecture and design specific. RealView Debugger menu options related to hardware breakpoints are grayed out if your target cannot support them.

Considerations when setting breakpoints

Be aware of the following when setting breakpoints:

- If your image is compiled with a high optimization level, then the effect of setting a breakpoint in the source view depends on where you set the breakpoint. For example, if you set a breakpoint on an inlined function, then a breakpoint is created for each instance of that inlined function. Therefore, if you set a hardware breakpoint on an inlined function, the target can run out of breakpoint resources.
- On some processors, a software breakpoint uses a hardware breakpoint resource, and all software breakpoints share this same resource. Therefore, if you run out of hardware breakpoint resource on those processors:
 - You can set a software breakpoint only if another software breakpoint is already set. Otherwise, the following error is displayed in the **Cmd** tab of the Output view:
Error V28507 (Vehicle): Insufficient hardware resources for software breakpoints.
 - When you attempt to set a hardware breakpoint, and a software breakpoint is already set, then a software breakpoint is set instead. In addition, the following message is displayed in the **Cmd** tab of the Output view:
Information: SW breakpoint set - unable to set HW breakpoint

- If the existing breakpoints are all hardware breakpoints, then you cannot set another breakpoint. if you attempt to set another breakpoint, the following error message is displayed in the **Cmd** tab of the Output view:
Error V28502 (Vehicle): No hardware resource available to set hardware breakpoint.
- Enabling a *Memory Management Unit* (MMU) might set a region of memory to read-only. If that memory region contains a software breakpoint, then that software breakpoint is set to read-only and cannot be removed. Therefore, make sure you clear software breakpoints before enabling the MMU.
- If RealView Debugger cannot set any more breakpoints, then rapid instruction step is used for high-level stepping. A message is also displayed to explain the type of step being used.
- When working with RVISS targets:
 - Watchpoints are available. These are called hardware breakpoints in RealView Debugger. You set these breakpoints using the same methods that are available for hardware development platforms. However, there is no corresponding hardware resource limit for RVISS targets.
 - Hardware breakpoints can use address ranges, data values, and data value range tests. They can also specify data size, processor mode, and pass counts.
 - Hardware breakpoints can be chained to form complex breakpoint conditions.

Note

Although you can set all types of hardware breakpoint on any RVISS target, the physical hardware might not support all the features of hardware breakpoints.

See also

- *Breakpoints in RealView Debugger* on page 1-36
- *Defining memory for a symmetric multiprocessor environment* on page 7-30
- *Breakpoints in different memory map regions* on page 11-8
- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints on custom memory mapped registers* on page 11-67
- Chapter 8 *Executing Images*
- the following in the *RealView Debugger RTOS Guide*:
 - Chapter 7 *Debugging Your OS Application*.

11.1.2 Breakpoint types for OS-aware connections

You can set thread breakpoints if you are debugging OS applications. The different types of thread breakpoints have a different icon. Table 11-1 shows the icons for thread breakpoint types.

Table 11-1 Breakpoint icons for OS-aware applications

Icon	Breakpoint/Tracepoint type
	Halted System Debug (HSD) breakpoint.
	Thread breakpoint in <i>Running System Debug</i> (RSD).
	RSD System breakpoint

See also

- *Breakpoints in RealView Debugger* on page 1-36
- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Chapter 8 Executing Images*
- the following in the *RealView Debugger RTOS Guide*:
 - Chapter 7 *Debugging Your OS Application*.

11.1.3 Unconditional and conditional breakpoints

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

Unconditional breakpoints

During execution, when the PC reaches the address where the breakpoint is set, the breakpoint is hit. An unconditional breakpoint always activates immediately it is hit, and always stops execution.

You can assign one or more actions to an unconditional breakpoint, which are performed when the breakpoint is activated.

Simple breakpoints

You can set an unconditional breakpoint:

- that performs the default action, which is to stop execution
- where RealView Debugger or your target hardware determines the type of breakpoint (that is, software or hardware).

This is called a *simple breakpoint*.

Note

Simple breakpoints are not added to the breakpoint history.

See also

- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Setting a simple breakpoint* on page 11-13
- *Breakpoints in RealView Debugger* on page 1-36
- *Events that determine when a breakpoint is hit* on page 1-36
- *Setting breakpoints from the breakpoint history list* on page 11-70
- Chapter 8 *Executing Images*
- Chapter 12 *Controlling the Behavior of Breakpoints*.

Conditional breakpoints

A breakpoint is conditional when you assign a condition or set of conditions that must be met for the breakpoint to be activated. A condition can be determined by:

- the result of C-style expression that equates to True or False
- the return value from a macro
- an event counter (for example, break on the fourth occurrence of reaching a particular address)
- a specific instance of a C++ object.

Note

Although hardware breakpoints can have conditions controlled in hardware, they are not identified as conditional by RealView Debugger. A breakpoint is identified as conditional by RealView Debugger only when the types of conditions listed here are assigned to that breakpoint.

A conditional breakpoint is activated when it is hit, and all specified conditions are met.

If you want to use multiple conditions, the order you specify the conditions affects the behavior of a breakpoint.

See also

- *Events that determine when a breakpoint is hit* on page 1-36
- *Breakpoints in RealView Debugger* on page 1-36
- *Conditional breakpoint activation* on page 1-37
- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Example of breakpoint behavior* on page 12-27
- Chapter 8 *Executing Images*
- Chapter 12 *Controlling the Behavior of Breakpoints*.

11.1.4 Breakpoints in different memory map regions

With memory mapping enabled, RealView Debugger sets a breakpoint based on the access rule for the memory at the chosen location. Table 11-2 shows the different types of memory that you can define in a memory map with RealView Debugger, and the default breakpoint type set by RealView Debugger.

Table 11-2 Breakpoint types for different memory map locations

Location	Breakpoint type set
RAM	Software
ROM	Hardware
Flash	Hardware
Write-only Memory (WOM)	Hardware
No Memory (NOM)	Hardware
Auto	Hardware

If you have to, you can manually set a hardware breakpoint in RAM.

If you attempt to set a software breakpoint at an address in Flash, ROM, WOM, NOM, or Auto memory, the operation fails by default. The error message displayed depends on whether or not memory mapping is enabled:

- Memory mapping enabled:
Error V004E (Vehicle): Memory map forbids software breakpoint at this address
- Memory mapping disabled:
Error V2801C (Vehicle): 0x050b0001: Unable to write sw breakpoint to memory.

However, you can use the failover qualifier to force the software breakpoint to be converted to a hardware breakpoint, depending on the available hardware resources. If the Debug Interface unit informs RealView Debugger that a hardware breakpoint is being used, RealView Debugger informs you that this has happened:

Information: Hardware breakpoint set - unable to set software breakpoint

If you attempt to set a breakpoint in a data or literals area of program memory within the **Disassembly** tab, the breakpoint is set and RealView Debugger issues the warning:

Warning: Breakpoint being set in data/literals of code.

See also

- *Breakpoints in RealView Debugger* on page 1-36
- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- Chapter 9 *Mapping Target Memory*
- Chapter 8 *Executing Images*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *BREAKINSTRUCTION* on page 2-55
 - Chapter 3 *RealView Debugger Predefined Macros*.

11.1.5 What happens when a breakpoint is activated?

When a breakpoint activates and program execution stops, RealView Debugger displays a message that identifies the type and location of the breakpoint, for example:

```
Stopped at 0x00008490 due to SW Instruction Breakpoint
Stopped at 0x00008490: DHRY_1\main Line 153
```

If you specify that execution is to continue after the breakpoint is activated, then these messages are not displayed.

If you want program execution to continue when a breakpoint is activated, and still be informed when this occurs, then:

1. Either:
 - assign a macro to the breakpoint that outputs appropriate messages and returns a value of zero
 - assign the message action to display an appropriate message.
2. Assign the continue command qualifier to the breakpoint.

See also

- *Breakpoints in RealView Debugger* on page 1-36
- *What happens when a breakpoint is activated?*
- *Breakpoint icons and color coding*
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Displaying user-defined messages when a breakpoint activates* on page 12-7
- *Setting the execution behavior for a breakpoint* on page 12-9
- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- *Example of breakpoint behavior* on page 12-27
- Chapter 8 *Executing Images*.

11.1.6 Breakpoint icons and color coding

Breakpoints are marked in the source-level and disassembly-level view in the gray margin, at the left side of the window. Standard software and hardware breakpoints are identified by a disc icon, which is color-coded to reflect the type and status of the breakpoint:

- | | |
|---------------|--|
| Red | A red icon shows that you have set an unconditional breakpoint, and that the breakpoint is enabled. |
| Yellow | A yellow icon shows that you have assigned a software condition to the breakpoint, and that the breakpoint is enabled. |
| White | A white icon shows that you have disabled an existing breakpoint. |

This color coding is also reflected in the Break/Tracepoints view.

If you try to set a breakpoint on a non-executable line, RealView Debugger places the breakpoint at the first line of executable code after your chosen location. If the lines preceding the breakpointed instruction are comments, declarations, or other non-executable code, they are marked with downward pointing arrows. Lines marked in this way are regarded as part of the breakpoint. Figure 11-1 on page 11-10 shows an example:

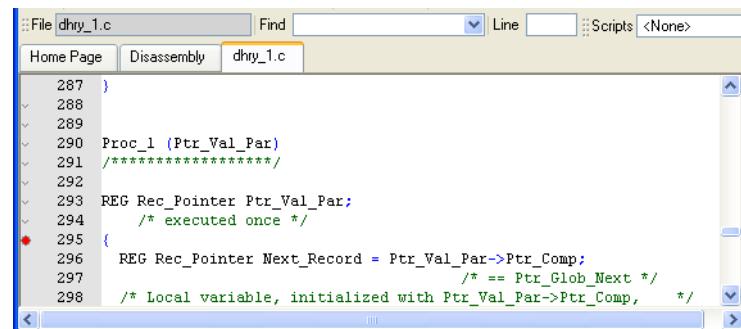


Figure 11-1 Breakpoint showing downward pointing arrows

See also

- [Breakpoints in RealView Debugger](#) on page 1-36
- [Software and hardware breakpoints](#) on page 11-3
- [Unconditional and conditional breakpoints](#) on page 11-6
- [What happens when a breakpoint is activated?](#) on page 11-9
- [Breakpoint icons and color coding](#) on page 11-9
- [Qualifying breakpoint line number references with module names](#)
- [Specifying address ranges](#)
- Chapter 8 [Executing Images](#).

11.1.7 Qualifying breakpoint line number references with module names

If you want to set a breakpoint in a source file that contains the line of code pointed to by the PC, you have only to specify a line number in that file. For example, using the CLI command BREAKINSTRUCTION you can set a breakpoint at line 92 in the dhry_1.c source file with:

```
breakinstruction #92
```

However, if you want to set a breakpoint in another source file associated with the loaded image, you must qualify the line number reference with the module name.

See also

- [Breakpoints in RealView Debugger](#) on page 1-36
- [Module naming conventions](#) on page 4-3
- [What happens when a breakpoint is activated?](#) on page 11-9
- [Breakpoint icons and color coding](#) on page 11-9
- [Qualifying breakpoint line number references with module names](#)
- [Specifying address ranges](#)
- Chapter 8 [Executing Images](#).

11.1.8 Specifying address ranges

Some processors support address ranges for hardware breakpoints, which halt the processor when the PC reaches any address in the specified address range. You specify an address range using either of the following formats:

start_addr..end_addr

Start address and an absolute end address, for example:

0x10000..0xFFFF

start_addr..+length

Start address and length of the address region, for example:

`0x10000..+0x1000`

These formats can be used in the various breakpoint dialog boxes available in the GUI, in addition to the CLI commands.

You can also use symbol names, such as macros, function names, and variables as the start address:

- `mymacro()..+1000`
- `main..+1000`
- `Arr_2_Glob..+64`

11.1.9 Help with specifying locations and values for hardware breakpoints

If your hardware supports it, RealView Debugger can assist you in specifying the location and value match entries when creating or editing a breakpoint. This enables you to:

- insert a string that specifies an address or value range
- insert an address or value mask
- insert a NOT address or value compare
- autocomplete a range, when you specify a symbol name.

These options are available from the right-arrow menu on the:

- Create Breakpoint dialog box
- Edit Breakpoint dialog box.

See also

- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9
- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Breakpoints in RealView Debugger* on page 1-36
- Chapter 8 *Executing Images*.

11.1.10 Breakpoints and image restarts

Restarting your program or reloading your image can have an effect on your breakpoints:

- To maintain your breakpoints when restarting your program:
 - reload the image using the Process Control view
 - set the PC to the image entry point.
- To clear your breakpoints when restarting your program:
 - unload the image, then load it again
 - reload the image from the Recent Images list

See also

- *Breakpoints in RealView Debugger* on page 1-36
- *What happens when a breakpoint is activated?* on page 11-9
- *Breakpoint icons and color coding* on page 11-9

- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- Chapter 4 *Loading Images and binaries*
- Chapter 8 *Executing Images*.

11.2 Setting a simple breakpoint

A simple breakpoint is a useful way to set a quick test point during a debugging session.

Note

The memory type where you set the breakpoint determines whether it is a software or hardware breakpoint.

See also:

- [Setting a breakpoint quickly](#)
- [Toggling a breakpoint at the current cursor position](#)
- [Setting a breakpoint with the Simple Break if X dialog box on page 11-14](#)
- [Effects of setting a breakpoint on page 11-14](#)
- [Simple breakpoints on page 11-6](#)
- [Breakpoints in different memory map regions on page 11-8](#)

11.2.1 Setting a breakpoint quickly

To set a simple breakpoint quickly:

1. Connect to your target.
2. Load the required image, for example `dhrystone.axf`.
3. Click the tab for the required code view.
For this example, click the `dhyr_1.c` tab to view the source file `dhyr_1.c`.
4. Locate the source line or instruction in your code view where you want to set the breakpoint. For example, line 158 in `dhyr_1.c`.
5. Double-click on the margin to the left of the required source line or instruction. A simple breakpoint is set, as indicated by the red icon in the margin. Figure 11-2 shows an example:

```

153     Int_2_Loc = 3;
154     strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
155     Enum_Loc = Ident_2;
156     Bool_Glob = ! Func_2 (Str_1_Loc, Str_2_Loc);
157     /* Bool_Glob == 1 */
158     while (Int_1_Loc < Int_2_Loc) /* loop body executed once */
159     {
160         Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
161         /* Int_3_Loc == 7 */
162         Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
163         /* Int_3_Loc == 7 */
164         Int_1_Loc += 1;

```

Figure 11-2 Breakpoint icon in the code view

See also

- [Breakpoint icons and color coding on page 11-9](#).

11.2.2 Toggling a breakpoint at the current cursor position

To toggle a breakpoint at the current cursor position, select **Debug** → **Breakpoints** → **Toggle Breakpoint at Cursor** from the Code window main menu.

11.2.3 Setting a breakpoint with the Simple Break if X dialog box

To set a breakpoint with the Simple Break if X dialog box:

1. Connect to your target.
2. Load the required image, for example dhystone.axf.
3. Click the tab for the required code view.
For this example, click the **dhry_1.c** tab to view the source file **dhry_1.c**.
4. Select **Debug → Breakpoints → Conditional → Break if X...** from the main menu. The Simple Break if X dialog box is displayed. Figure 11-3 shows an example:

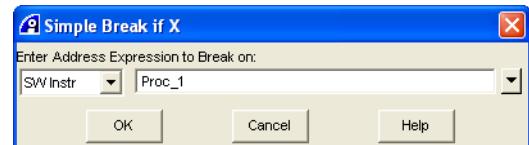


Figure 11-3 Simple Break if X dialog box

5. Select the breakpoint type for this breakpoint. The default is **SW Instr**.
6. Enter the address expression for the breakpoint. This can be:
 - a specific line number in the source code, with or without a module name prefix
 - a specific address, which can be the address of a variable or function
 - a macro that returns an address
 - an address range
 - the start of a function
 - a function entry point.
 Alternatively, click the expression selector button, to select from:
 - various lists, including your Favorites List
 - expressions used in previous breakpoints during this session.
 For example, enter **Proc_5** to set a breakpoint at the start of the **Proc_5()** function.
7. Click **OK** to set the breakpoint and close the dialog box.

See also

- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Setting a breakpoint at the start of a function* on page 11-39
- *Setting a breakpoint at the entry point to a function* on page 11-40.

11.2.4 Effects of setting a breakpoint

Setting a breakpoint updates the Break/Tracepoints view, if it is visible, and the Output view shows the CLI command used to set the breakpoint:

- If you set a breakpoint in a source file tab, the command has the following format:
`binstr \MODULE\#line_number:char_pos`
For example, `binstr \DHRY_1\#158:3`.
- If you set a breakpoint in the **Disassembly** tab, the command has the following format:
`binstr address`

For example, `binstr 0x000084D0`.

Effect of memory type on breakpoints

The type of breakpoint set, either software or hardware, is determined by the memory type:

- Setting breakpoints in source-level view inserts a software instruction breakpoint by default. This is set using a `BREAKINSTRUCTION` command. RealView Debugger attempts to set a software breakpoint if the code is in RAM.
- If your code is in ROM or Flash, RealView Debugger sets a hardware breakpoint if one is available. This is set using a `BREAKEXECUTION` command. An error message is displayed if no such breakpoint is available.
- If you attempt to set a software breakpoint at an address that you know is in RAM, and RealView Debugger unexpectedly sets a hardware breakpoint or fails, then check the **Memory Map** tab to see if memory mapping is enabled. If memory mapping is enabled, make sure the memory area of interest is defined as RAM. RealView Debugger always sets a hardware breakpoint in a memory area that is marked as Default Mapping.

— **Note** —

In some instances, RealView Debugger fails to set a software breakpoint. This situation can arise when memory mapping is disabled, because RealView Debugger attempts to treat all memory as RAM by default. You can use the `failover` qualifier with the `BREAKINSTRUCTION` command to force the software breakpoint to be converted to a hardware breakpoint.

See also

- *Creating a temporary memory map entry* on page 9-18
- *Breakpoints in different memory map regions* on page 11-8
- *Viewing breakpoint information* on page 11-20
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the `BREAKEXECUTION` and `BREAKINSTRUCTION` commands.

11.3 Setting an unconditional breakpoint with specific attributes

To set an unconditional breakpoint with specific attributes, you must use the Create Breakpoint dialog box.

To set an unconditional breakpoint with specific attributes:

1. For a software or hardware instruction breakpoint, locate the line of source or disassembly where you want to set the breakpoint.
For a hardware data breakpoint, any line of source or disassembly is suitable.
2. Right-click in the margin of the source or disassembly view, as appropriate, to display the context menu.
3. Select **Create Breakpoint...** from the context menu to display the Create Breakpoint dialog box. Figure 11-4 shows an example. The chosen location is inserted in the Location field.

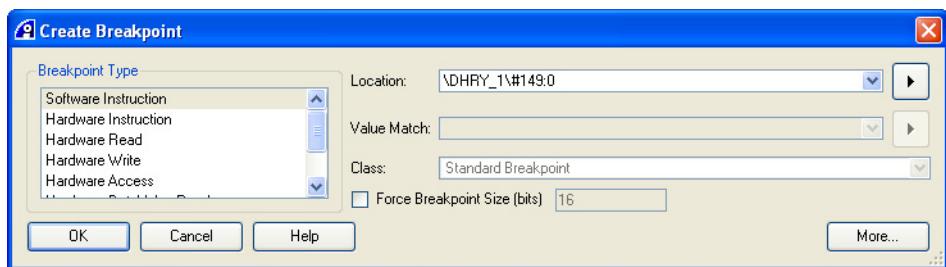


Figure 11-4 Create Breakpoint dialog box

4. Select the breakpoint type for this breakpoint. The default is **Software Instruction**. For hardware targets, the following breakpoint types are listed:
 - **Hardware DataValue Access**
 - **Hardware DataValue Read**
 - **Hardware DataValue Write**.
 If you select any of these types, skip the next step.
5. Change the address expression for the breakpoint, if required. This can be:
 - a specific line number in the source code, with or without a module name prefix
 - a specific address, which can be the address of a variable or function
 - a macro that returns an address
 - an address range
 - the start of a function.
 - a function entry point.
 For example, enter **Proc_5** to set a breakpoint at the start of the **Proc_5()** function.
6. Do the following depending on the breakpoint type you selected:
 - If you selected **Software Instruction** as the breakpoint type, optionally select **Force Breakpoint Size (bits)**. When you select **Force Breakpoint Size (bits)**, the entry field is enabled for you to specify the size in bits. Normally, the breakpoint size is selected automatically using the debug information that is currently loaded on the target. However, if you are trying to set a Thumb breakpoint on a target where there is no debug information available, you must use this option to force the breakpoint size.

- If you selected **Hardware Access**, **Hardware Read**, or **Hardware Write**, optionally, enter a value in the Value Match field.
 - If you selected **Hardware DataValue Access**, **Hardware DataValue Read**, or **Hardware DataValue Write**, you must enter a value in the Value Match field.
7. Click **OK** to set the breakpoint and close the dialog box.

See also:

- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Clearing breakpoints* on page 11-18
- *Disabling a breakpoint* on page 11-22
- *Enabling a breakpoint* on page 11-23
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Setting breakpoints on lines of source code* on page 11-34
- *Setting breakpoints on instructions* on page 11-37
- *Setting breakpoints on functions* on page 11-39
- *Setting a breakpoint at the start of a function* on page 11-39
- *Setting a breakpoint at the entry point to a function* on page 11-40
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Forcing the size of a software breakpoint* on page 11-62
- *Specifying processor exceptions (global breakpoints)* on page 11-65
- *Setting breakpoints on custom memory mapped registers* on page 11-67
- Chapter 12 *Controlling the Behavior of Breakpoints*.

11.4 Clearing breakpoints

You can clear breakpoints individually, or in a single operation.

See also:

- *Clearing a breakpoint quickly*
- *Clearing specific breakpoints*
- *Clearing all breakpoints* on page 11-19
- *Considerations when clearing breakpoints* on page 11-19.

11.4.1 Clearing a breakpoint quickly

To clear a breakpoint quickly when it is visible in the code view or disassembly view, double-click on the marker icon in the margin. The breakpoint is removed.

— Note —

Where you have set multiple breakpoints, for example on a multi-statement line, clearing a breakpoint this way removes only one of the breakpoints.

11.4.2 Clearing specific breakpoints

To clear specific breakpoints:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view, if it is not already visible. Figure 11-5 shows an example:

The screenshot shows a Windows-style dialog box titled "Break/Tracepoints". It contains a table with two columns: "Type" and "Value". There are three entries, each preceded by a plus sign (+) and a red circular icon with a white dot (indicating a breakpoint). The first entry is "Instr" with a value of "0x000081E4". The second entry is "Instr" with a value of "0x0000850C". The third entry is "Instr" with a value of "0x00008FA8". At the bottom of the window, there is a status bar displaying "ARM7TDMI@RVISS".

Type	Value
+ ● Instr	0x000081E4
+ ● Instr	0x0000850C
+ ● Instr	0x00008FA8

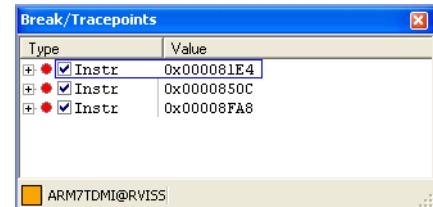
Figure 11-5 Breakpoint in the Break/Tracepoints view

2. Right-click on the breakpoint you want to clear to display the context menu.
3. Select **Delete** from the context menu. The breakpoint is cleared and its details removed from the Break/Tracepoints view.
4. Repeat these steps for any other breakpoints that you want to clear.

11.4.3 Clearing all breakpoints

To clear all breakpoints:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view, if it is not already visible. Figure 11-6 shows an example:



The screenshot shows a Windows-style dialog box titled "Break/Tracepoints". It has two columns: "Type" and "Value". There are three entries, each preceded by a red circular icon with a white dot and a checkmark. The "Value" column contains memory addresses: 0x000081E4, 0x0000850C, and 0x00008FA8. At the bottom of the window, there is a status bar with the text "ARM7TDMI@RVISS".

Type	Value
+ ● <input checked="" type="checkbox"/> Instr	0x000081E4
+ ● <input checked="" type="checkbox"/> Instr	0x0000850C
+ ● <input checked="" type="checkbox"/> Instr	0x00008FA8

Figure 11-6 Breakpoint in the Break/Tracepoints view

2. Right-click in the view to display the context menu.
3. Select **Delete All** from the context menu.

— Note —

Be aware that this also clears any tracepoints that you have set. If you want to clear breakpoints and also leave any tracepoints set, follow the steps described in *Clearing specific breakpoints* on page 11-18.

11.4.4 Considerations when clearing breakpoints

If the breakpoint is part of a breakpoint chain, then:

- when you clear the first breakpoint unit in the chain, all breakpoint units in the chain are cleared
- when you clear any other breakpoint unit in the chain, only that breakpoint unit is cleared.

See also

- *Chaining hardware breakpoints* on page 11-63.

11.5 Viewing breakpoint information

You can view information about the breakpoints you set, such as the CLI commands used and any conditions or actions assigned to them.

See also:

- *Breakpoints in the Break/Tracepoints view*
- *Viewing the breakpoint details* on page 11-21.

11.5.1 Breakpoints in the Break/Tracepoints view

The Break/Tracepoints view shows entries in a tree view giving the Type and Value of each breakpoint you set. Figure 11-7 shows an example:

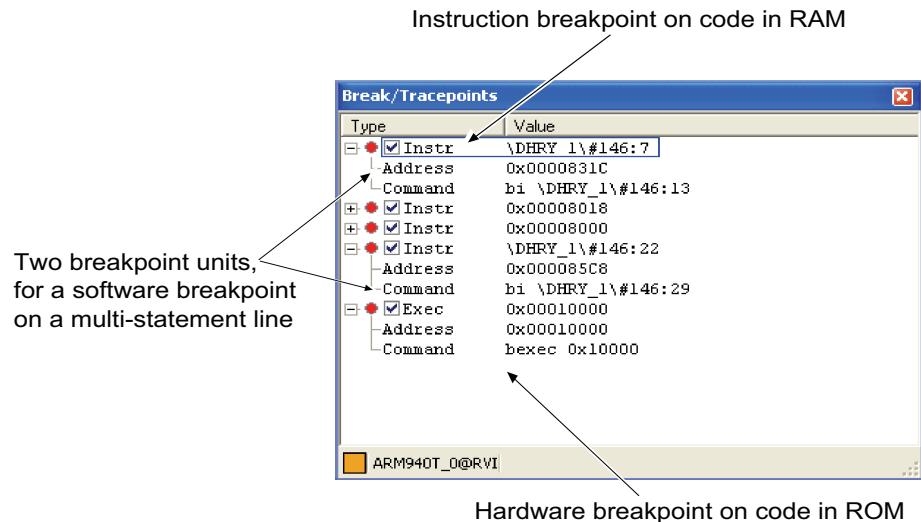


Figure 11-7 Breakpoints in the Break/Tracepoints view

11.5.2 Viewing the breakpoint details

To view details for a breakpoint:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view, if it is not already visible. Figure 11-8 shows an example:

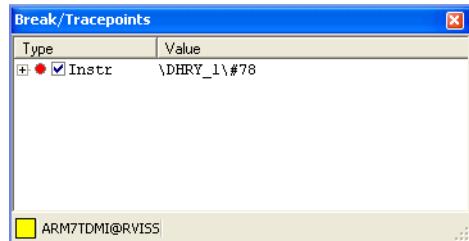


Figure 11-8 Breakpoint in the Break/Tracepoints view

2. Expand the entry for the breakpoint. This shows the location of the breakpoint, and the CLI command that was used to set it. Figure 11-9 shows an example:

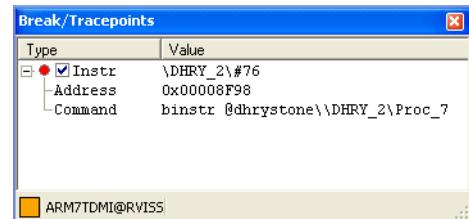


Figure 11-9 Breakpoint entry expanded

3. Right-click on the entry for the breakpoint to display the context menu.
4. Select **Properties** from the context menu. The full breakpoint details are displayed. Figure 11-10 shows an example:

If any conditions or actions are assigned to the breakpoint, then these are also listed.

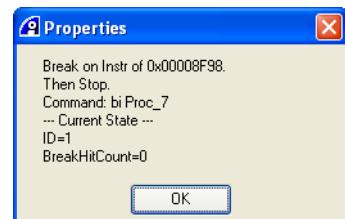


Figure 11-10 Full breakpoint details

See also

- Chapter 12 *Controlling the Behavior of Breakpoints*.

11.6 Disabling a breakpoint

If you want to prevent a breakpoint from being activated, but still keep the breakpoint available, you can disable the breakpoint.

See also:

- *Disabling a breakpoint at the current cursor position*
- *Disabling a specific breakpoint*
- *Considerations when disabling breakpoints.*

11.6.1 Disabling a breakpoint at the current cursor position

To disable a breakpoint at the current cursor position in a code view, select **Debug** → **Breakpoints** → **Enable/Disable Break at Cursor** from the Code window main menu.

The color of the breakpoint icon in the code view changes to a white disk to show that this breakpoint is disabled. You can still view the breakpoint in the source file tab or **Disassembly** tab, and in the Break/Tracepoints view.

See also

- *Breakpoint icons and color coding* on page 11-9.

11.6.2 Disabling a specific breakpoint

To disable a specific breakpoint:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoint view.
2. Deselect the check box for the breakpoint that you want to disable.

The color of the breakpoint icon in the code view changes to a white disk to show that this breakpoint is disabled. You can still view the breakpoint in the source file tab or **Disassembly** tab, and in the Break/Tracepoints view.

See also

- *Breakpoint icons and color coding* on page 11-9.

11.6.3 Considerations when disabling breakpoints

If the breakpoint is part of a breakpoint chain, then:

- when you disable the first breakpoint unit in the chain, all breakpoint units in the chain are disabled
- when you disable any other breakpoint unit in the chain, that breakpoint unit and all subsequent breakpoint units in the chain are disabled.

See also

- *Chaining hardware breakpoints* on page 11-63.

11.7 Enabling a breakpoint

Breakpoints are automatically enabled when you create them. However, if you have disabled a breakpoint, you can re-enable the breakpoint.

See also:

- [Enabling a breakpoint in the code view](#)
- [Enabling a specific breakpoint](#)
- [Considerations when enabling breakpoints.](#)

11.7.1 Enabling a breakpoint in the code view

To enable a breakpoint in the code view:

1. Locate the line of source or disassembly containing the disabled breakpoint.
2. Right-click on the breakpoint icon to display the context menu.
3. Select **Enable Breakpoint** from the context menu.

The color of the breakpoint icon in the code view changes to the relevant color for that breakpoint, to show that this breakpoint is enabled.

— Note —

If no breakpoint exists at the cursor position, a new simple breakpoint is created.

See also

- [Simple breakpoints](#) on page 11-6
- [Breakpoint icons and color coding](#) on page 11-9.

11.7.2 Enabling a specific breakpoint

To enable a specific breakpoint:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoint view.
2. Select the check box for the breakpoint that you want to enable.

The color of the breakpoint icon in the code view changes to the relevant color for that breakpoint, to show that this breakpoint is enabled.

See also

- [Breakpoint icons and color coding](#) on page 11-9.

11.7.3 Considerations when enabling breakpoints

If the breakpoint is part of a breakpoint chain, then:

- when you enable the first breakpoint unit in the chain, all breakpoint units in the chain are enabled
- when you disable or enable any other breakpoint unit in the chain, that breakpoint unit and all subsequent breakpoint units in the chain are disabled or enabled.

See also

- *Chaining hardware breakpoints* on page 11-63.

11.8 Editing a breakpoint

You can edit an existing breakpoint to change its type, location, conditions, or associated actions.

Note

You cannot edit a breakpoint that is part of a breakpoint chain. If you do, RealView Debugger displays the following error message:

Error: This breakpoint cannot be edited, or copied, as it is chained to another breakpoint.

See *Chaining hardware breakpoints* on page 11-63 for more details on chained breakpoints.

To edit a breakpoint:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view, if it is not already visible. Figure 11-11 shows an example:

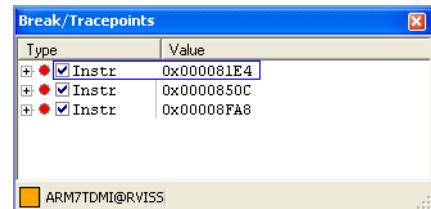


Figure 11-11 Breakpoint in the Break/Tracepoints view

2. Right-click on the breakpoint you want to edit to display the context menu.
3. Select **Edit...** from the context menu to display the Edit Breakpoint dialog box. Figure 11-12 shows an example:

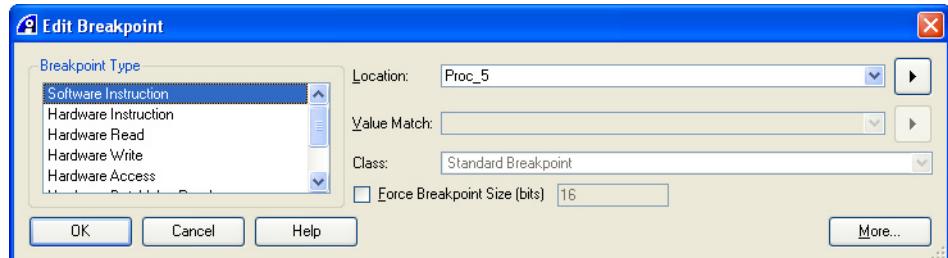


Figure 11-12 Edit Breakpoint dialog box

4. Make the required changes to the breakpoint. You can:
 - Change the Breakpoint Type.
 - Change the Location of the breakpoint.
 - Specify a data value to match against if you select one of the hardware access types.
 - Force the size of a software breakpoint.

When you select Force Breakpoint Size (bits), the entry field is enabled for you to specify the size in bits. Normally, the breakpoint size is selected automatically using the debug information that is currently loaded on the target. However, if you are trying to set a Thumb® breakpoint on a target where there is no debug information available, you must use this option to force the breakpoint size.

Note

This option is disabled for hardware breakpoints.

- Click **More...** to display conditions and actions that you can assign.

For this example, change the Location to enter `Proc_5\@entry`, to set a breakpoint on the entry point to the `Proc_5` function.

5. Click **OK** to set the breakpoint.
6. If you change the Location or Breakpoint Type, a prompt dialog box is displayed to ask if you want to replace the current breakpoint:
 - Click **Yes** to set the breakpoint, and replace the existing one.
 - Click **No** to keep the existing breakpoint, and set a new breakpoint.

Note

The breakpoint command includes the `modified` command qualifier, to show that you have edited the command.

See also:

- *Help with specifying locations and values for hardware breakpoints*
- Chapter 12 *Controlling the Behavior of Breakpoints*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.8.1 Help with specifying locations and values for hardware breakpoints

If supported by your hardware, RealView Debugger provides the following menus to help you specify locations and values for hardware breakpoint:

Address Range and Mask menu

Click the right-arrow at the side of the Location data field to display the Address Range and Mask menu. Use options from this menu to specify an expression range or mask a group of instructions.

Value Range and Mask menu

Click the right-arrow at the side of the Value Match data field to display the Value Range and Mask menu. Use options from this menu to test a range of values or mask a range of data values.

Also, the contents of these menus depend on your target.

Choose from the available options to set up your hardware breakpoint:

Range options

Enter the start address, or data value, for the breakpoint then select this option to specify a range, for example the address range `0x800FF..0x80A00`. The separators “`..`” are automatically inserted for you.

Range by Length options

Enter the start address, or data value, for the breakpoint then select this option to specify a range by length, for example the address range `0x800FF..+0x1111`. The separators “`..+`” are automatically inserted for you.

Mask options

Enter the address, or data value, for the breakpoint then select this option to specify a mask. RealView Debugger inserts the mask for you, for example `0x800FF $MASK$=0xFFFF`. Change this mask as required.

The mask is a bitwise AND mask applied to the address or data value being compared, before matching it against the one you specified. For example, if you specify the location `0x0111` and the mask `0xFFFF`, then addresses such as `0x0111`, `0x10111` and `0xFFC70111` match successfully.

The breakpoint activates when the address, or data value, matches the given value after masking.

NOT Compare options

Enter the address, or data value, for the breakpoint then select this option to specify a NOT operation, for example `NOT 0x800FF`.

Similarly, use this option to specify a range of addresses, or data values, to ignore, for example `NOT 0x0500..+0x0100`.

Autocomplete Range options

Enter a symbol and then select this option to compute the end-of-range address based on the symbol size:

- If you enter a function name then the autocompleted range is from the start to the end of the function, for example, `Proc_5..+0x1C`.
- If you enter a global variable the autocompleted range is the variable storage address plus the variable size, for example, `Arr_2_Glob..+0x2710`.

Note

You cannot mix ranges and masks.

See also

- *Specifying address ranges* on page 11-10
- Chapter 12 *Controlling the Behavior of Breakpoints*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.9 Copying a breakpoint

By copying a breakpoint, you can create a new breakpoint based on the properties of an existing breakpoint.

Note

You cannot copy a breakpoint that is part of a breakpoint chain. If you do, RealView Debugger displays the following error message:

Error: This breakpoint cannot be edited, or copied, as it is chained to another breakpoint.

To copy a breakpoint:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view, if it is not already visible. Figure 11-13 shows an example:

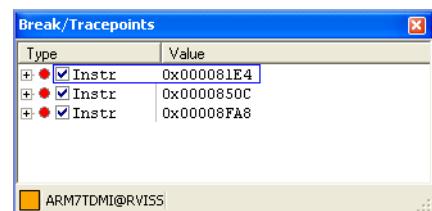


Figure 11-13 Breakpoint in the Break/Tracepoints view

2. Right-click on the breakpoint you want to copy to display the context menu.
3. Select **Edit...** from the context menu to display the Edit Breakpoint dialog box. Figure 11-14 shows an example:

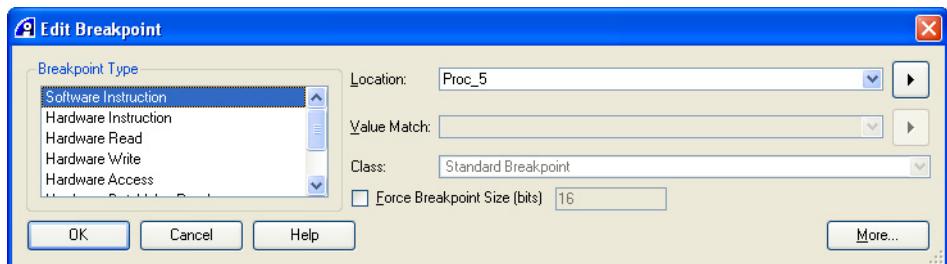


Figure 11-14 Edit Breakpoint dialog box

4. Change either the Breakpoint Type or the Location of the breakpoint. For this example, change the Location to **Proc_5**, to set a new breakpoint at the first location of the **Proc_5** function.
5. Make any other changes to the breakpoint that you require. You can:
 - Change the Breakpoint Type.
 - Specify a data value to match against if you selected one of the hardware access types.
 - Force the size of a software breakpoint.

When you select Force Breakpoint Size (bits), the entry field is enabled for you to specify the size in bits. Normally, the breakpoint size is selected automatically using the debug information that is currently loaded on the target. However, if you are trying to set a Thumb breakpoint on a target where there is no debug information available, you must use this option to force the breakpoint size.

Note

This option is disabled for hardware breakpoints.

- Click **More...** to display conditions and actions that you can assign.
6. Click **OK** to set the breakpoint. A prompt dialog box is displayed to ask if you want to replace the current breakpoint.
 7. Click **No** to keep the existing breakpoint, and set a new breakpoint.

See also:

- *Help with specifying locations and values for hardware breakpoints* on page 11-26
- *Chaining hardware breakpoints* on page 11-63
- Chapter 12 *Controlling the Behavior of Breakpoints*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.10 Finding a breakpoint in the code view

You can quickly locate a software instruction (Instr) or hardware execution (Exec) breakpoint in your code.

To view the location where a breakpoint is set:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view, if it is not already visible. Figure 11-15 shows an example:

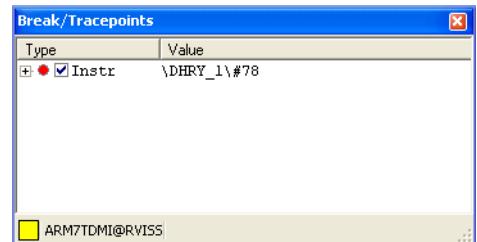


Figure 11-15 Breakpoint in the Break/Tracepoints view

2. Right-click on the required breakpoint in the Break/Tracepoints view to display the context menu.
3. Select **View Code** from the context menu. The code view changes to show the location of the breakpoint:
 - If the breakpoint was set using an address reference, then the location is shown in the **Disassembly** tab. Figure 11-16 shows an example.
 - If the breakpoint was set using a source line reference, then the location is shown in the source file tab.

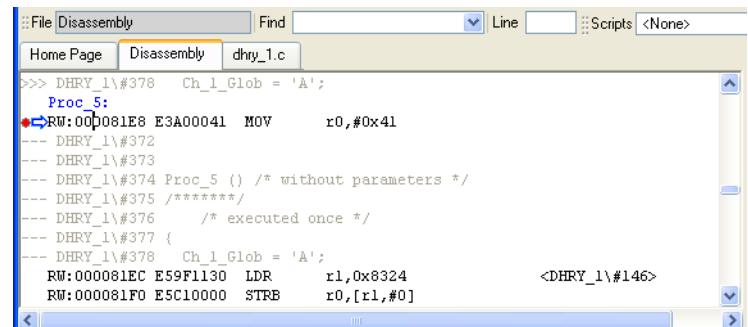


Figure 11-16 Locating a breakpoint

11.11 Viewing the target hardware breakpoint support

Your target hardware has a limit on the number and complexity of the hardware breakpoints that you can set. RealView Debugger enables you to determine the hardware breakpoint support that is available on your target hardware.

To see the hardware breakpoint support for your target hardware:

1. Connect to your target hardware.
2. Select **Debug → Breakpoints → Hardware → Show Break Capabilities of HW...** from the Code window main menu to display the information dialog box describing the support available for your target processor. Figure 11-17 shows an example:

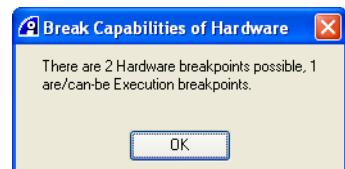


Figure 11-17 Example hardware break characteristics

The details shown depends on the target processor and the Debug Interface used to make the connection. Figure 11-17 shows the details for an ARM940T™ using DSTREAM or RealView ICE.

— Note —

RealView Debugger can reserve one breakpoint unit for internal use. Therefore, there might be one less breakpoint available to you. An error message is displayed if you try to set a hardware breakpoint when the limit is reached.

See also:

- *Connecting to a target* on page 3-27
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63.

11.12 Setting breakpoints by dragging and dropping

You can set breakpoints by dragging and dropping program items from your source code onto the Break/Tracepoints view.

To set a breakpoint by dragging and dropping a program item:

1. Select **Break/Tracepoints** from the **View** menu to display the Break/Tracepoints view. Figure 11-18 shows an example:



Figure 11-18 Break/tracepoint view

2. Locate the program item in your source code, such as a function call. Figure 11-19 shows an example source code view for the dhystone.axf image.

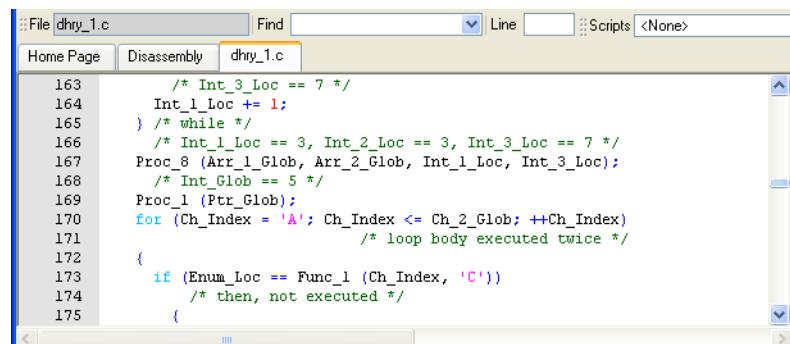


Figure 11-19 Source code view

3. Double-click on the program item to highlight it. In the example shown in Figure 11-19, double-click on the Proc_1 function name.
4. Drag the item and drop it onto the Break/Tracepoints view. The breakpoint type selection dialog box is displayed. Figure 11-20 shows an example:

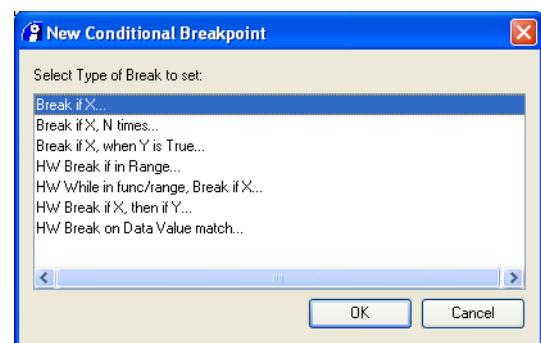


Figure 11-20 Breakpoint type selection dialog box

5. Select the breakpoint type you want to set. The available options depend on the target:
 - **Break if X...** displays the Simple Break if X dialog box

- **Break if X, N Times...** displays the Simple Break if X, N Times dialog box
- **Break if X, when Y is True...** displays the Simple Break if X, when Y is True dialog box
- **HW Break if in Range...** displays the HW Break if in Range dialog box
- **HW While in func/range, Break if X...** displays the HW While in func/range, Break if X dialog box
- **HW Break if X, then if Y...** displays the HW Break if X, then if Y dialog box.
- **HW Break on Data Value match...** displays the HW Break on Data Value match dialog box.

In this example, leave the default type selected, that is **Break if X....**

6. In the dialog box that is displayed, modify the breakpoint parameters as required. In this example, change the type of breakpoint from **HW Instr** to **SW Instr** in the Simple Break if X dialog box.
7. Click **OK** to set the breakpoint and close the dialog box.

In this example, a software breakpoint is set at the start of the function `Proc_1`.

See also:

- *Setting a breakpoint with the Simple Break if X dialog box* on page 11-14
- *Breaking on accesses at multiple locations* on page 11-52
- *Setting breakpoints for location-specific data values* on page 11-54
- *Breaking on a data value match within a range of addresses* on page 11-56
- *Breaking in a function or range that also depends on a data value* on page 11-56
- *Setting a breakpoint that activates after a number of passes* on page 12-13
- *Setting a breakpoint that depends on the result of an expression* on page 12-18.

11.13 Setting breakpoints on lines of source code

You can set only a single breakpoint of the same type (software or hardware) on each statement in your code. Lines of code can contain a single statement, or multiple statements (such as a for loop).

See also:

- *Setting a breakpoint on a single-statement line*
- *Setting breakpoints on a multi-statement line* on page 11-35.

11.13.1 Setting a breakpoint on a single-statement line

To set a breakpoint on a single-statement line:

1. Connect to a suitable target.
2. Load the image to the target.
3. Open the source file containing the line of source where you want to set the breakpoint.
4. Double-click in the margin, that is the gray area to the left of a line, to set a simple breakpoint quickly. Figure 11-21 shows an example. You can also double-click on the line number if this is visible.

```

File dhy_1.c Find Line Scripts <None>
Home Page Disassembly dhy_1.c
108 printf ("\n");
109 printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
110 printf ("\n");
111 if (Reg)
112 {
113     printf ("Program compiled with 'register' attribute\n");
114     printf ("\n");
115 }
116 else
117 {
118     printf ("Program compiled without 'register' attribute\n");
119     printf ("\n");
120 }

```

Figure 11-21 Setting an unconditional breakpoint on a line

This example sets a default software breakpoint marked by a red disc in the margin.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- *Opening source files for a loaded image* on page 4-26.

11.13.2 Setting breakpoints on a multi-statement line

To set breakpoints on a multi-statement line:

1. Connect to a suitable target.
2. Load the image to the target.
3. Open the source file containing the line of source where you want to set the breakpoint, for example, dhry_1.c.
4. In the dhry_1.c source, locate the for loop at line 145.
5. Right-click on the first statement, that is for (Run_Index = 1.
6. Select **Insert Breakpoint on Statement** from the context menu.
7. Right-click on the middle statement, that is Run_Index <= Number_Of_Runs.
8. Select **Insert Breakpoint on Statement** from the context menu.
9. Right-click on the last statement, that is ++Run_Index.
10. Select **Insert Breakpoint on Statement** from the context menu.

Although you have set three breakpoints, because they are associated with the same line of code, only one breakpoint indicator is visible in the source tab.

Effect of optimizations

Be aware that optimizations might affect whether or not you can set a breakpoint on each statement of a multi-statement line. For example:

- The prebuilt image dhystone.axf is compiled with -O0, giving the best debug view. This enables you to set a breakpoint on each statement as described in the procedure.
- If you rebuild the image with compiler optimization levels -O1 and -O2, you can still set a breakpoint on each statement of a multi-statement line. However, the **Insert Breakpoint on Statement** context menu option is not available for the first statement. You must use **Insert Breakpoint** instead, then use **Insert Breakpoint on Statement** for the remaining statements.
- If you rebuild the image with compiler optimization levels -O3, then you cannot set a breakpoint on each statement of a multi-statement line. You can use **Insert Breakpoint** to set a breakpoint only on one of the statements.

Managing multiple breakpoints on a multi-statement line

If you have set multiple breakpoints on a source line containing multiple statements or on inline functions, then disabling the last breakpoint changes the marker disc. Figure 11-22 on page 11-36 shows an example. If you disable any other breakpoint on the line, the marker remains unchanged.

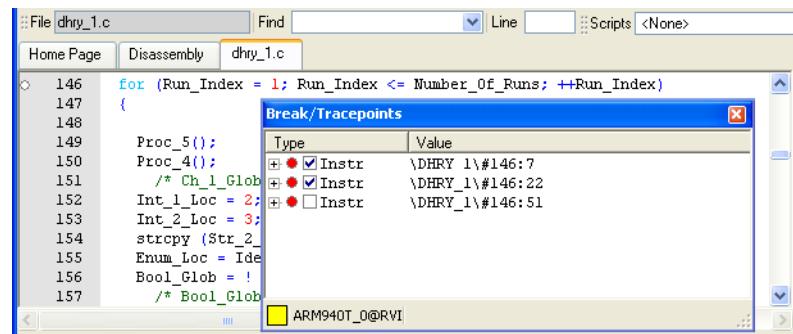


Figure 11-22 Multiple breakpoints on a multi-statement line

You cannot place two breakpoints of the same type on the same statement. Also, you cannot set breakpoints on lines marked with downward pointing arrows.

See also

- *Viewing breakpoint information* on page 11-20.
- *ARM® Compiler toolchain Compiler Reference*

11.14 Setting breakpoints on instructions

You can set breakpoints on instructions in the **Disassembly** tab.

See also:

- *Setting a breakpoint on an instruction*
- *Setting a breakpoint at the destination of a branch instruction.*

11.14.1 Setting a breakpoint on an instruction

To set a breakpoint on an instruction:

1. Connect to a suitable target.
2. Load the image to the target.
3. Click the **Disassembly** tab to show the disassembly view.
4. Locate the instruction where you want to set the breakpoint.
5. Right-click on the instruction to display the context menu.
6. Select **Insert Breakpoint** from the context menu.

The breakpoint is set at the address of the instruction.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4.

11.14.2 Setting a breakpoint at the destination of a branch instruction

To set a breakpoint on the destination of a branch instruction:

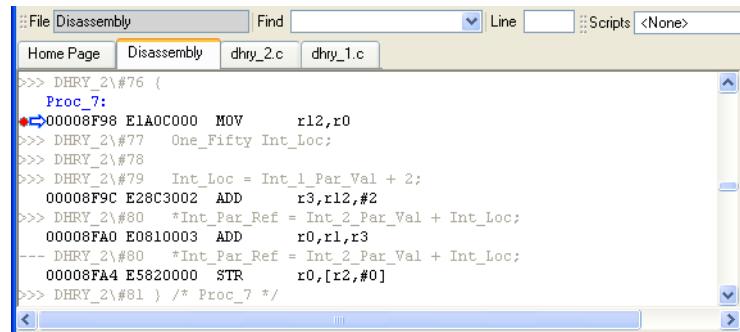
1. Connect to a suitable target.
2. Load the image to the target.
3. Click the **Disassembly** tab to show the disassembly view.
4. Locate the required branch instruction in the **Disassembly** tab.
5. Right-click on the destination address specified in the branch instruction to display the context menu.
6. Select **Create Breakpoint...** from the context menu. The Create Breakpoint dialog box is displayed.
7. Enter the destination address for the branch instruction in the Location field. For example, enter **Proc_7**.
8. Click **OK**. A breakpoint is set on the instruction at the specified branch address (0x8F90 in this example).

The Break/Tracepoints view is updated with the new breakpoint (if visible) and the Output view shows the breakpoint command:

```
binstr Proc_7
```

9. To see the breakpoint at the branch destination:
 - a. Right-click on the destination address again to display the context menu.

- b. Select **Locate Address...** in the context menu.
- c. Click **Set**. The location of the branch destination is displayed. Figure 11-23 shows an example:



```

File Disassembly Find Line Scripts <None>
Home Page Disassembly dhry_2.c dhry_1.c
>>> DHRY_2\#76 {
    Proc_7:
    <00008F98 E1AOC000 MOV      r12,r0
>>> DHRY_2\#77  One_Fifty Int_Loc;
>>> DHRY_2\#78
>>> DHRY_2\#79  Int_Loc = Int_1_Par_Val + 2;
    00008F9C E28C3002 ADD      r3,r12,#2
>>> DHRY_2\#80  *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
    00008FA0 E0810003 ADD      r0,r1,r3
--- DHRY_2\#80  *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
    00008FA4 E5820000 STR      r0,[r2,#0]
>>> DHRY_2\#81 } /* Proc_7 */

```

Figure 11-23 Setting an unconditional breakpoint on an instruction

As with the source-level view, RealView Debugger sets a software or hardware breakpoint depending on where your program is stored and what breakpoints are available.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- *Viewing breakpoint information* on page 11-20.

11.15 Setting breakpoints on functions

You can set a breakpoint in a function, without having to locate the source for the function. You can do this from any source line that contains a call to the function.

See also:

- *Setting a breakpoint at the start of a function*
- *Setting a breakpoint at the entry point to a function* on page 11-40
- *Setting a breakpoint in an inlined function* on page 11-41
- *Setting a breakpoint in a C++ template* on page 11-42
- *Setting a breakpoint at the start of a member function in a C++ class* on page 11-43
- *Setting a breakpoint at the location of a Call Stack entry* on page 11-43
- *Setting breakpoints on functions from the Function/Label list* on page 11-45.

11.15.1 Setting a breakpoint at the start of a function

To set a breakpoint at the start of a function:

1. Select **Symbols** from the **View** menu. The Symbols view is displayed.
2. Click the **Functions** tab to display the list of functions.
3. Locate the function where you want to set the breakpoint, for example, Func_2.
4. Right-click on the function name to display the context menu.
5. Select **Insert Breakpoint** from the context menu. The breakpoint is set at the start of the selected function.

Viewing the function where you set a breakpoint

To view the function where you set the breakpoint:

1. Right-click on the function name again (Func_2 in the previous example) to display the context menu.
2. Select the required option from the context menu:

View Disassembly

RealView Debugger displays the **Disassembly** tab to show the disassembly for the function.

View Source

RealView Debugger opens the source dhry_2 at the function Func_2, and shows that the breakpoint is set at line 143.

11.15.2 Setting a breakpoint at the entry point to a function

You can set a breakpoint at the entry point of a function by qualifying the function name with the @entry symbol. This identifies the first location in the function after the code that sets up the function parameters. In general, *function*\@entry refers to either:

- the first executable line of code in that function
- the first local variable that is initialized in that function.

To set a breakpoint on the entry point to a function:

1. Locate a source line containing a call to the function. For example locate line 156 in dhry_1.c.
2. Right-click on the function name, for example, Func_2, to display the context menu.
3. Select **Create Breakpoint...** from the context menu to display the Create Breakpoint dialog box.
4. Enter *function*\@entry for the breakpoint Location. In this example, enter **Func_2\@entry**.
5. Click **OK**. The breakpoint is set in the body of the function.

If no lines exist that set up any parameters for the function (for example, an embedded assembler function), then the following error message is displayed:

Error: E0039: Line number not found.

As an example, if you have a function Func_1(value) you might want to set a breakpoint that activates only when the argument value has a specific value on entry to the function:

`binstr,when:{value==2} Func_1\@entry`

Note

This is different to specifying a function name without the @entry qualifier. For example, in the dhystone.axf image:

- specifying Func_1 sets a breakpoint at 0x00009044, the first address in the function
- specifying Func_1\@entry sets a breakpoint at 0x0000904C, which is the first line of executable code of the function (line 122 in dhry_2.c).

11.15.3 Setting a breakpoint in an inlined function

Setting a breakpoint in a function that is inlined might result in the breakpoint being set at multiple locations, depending on the breakpoint type.

Setting software breakpoints in inlined functions

If you set a software breakpoint in an inlined function in RAM, then be aware that the breakpoint is set on all locations where the function is called. Software breakpoints are set on the locations of the function calls in the order they are found by RealView Debugger. A message is displayed informing you of how many breakpoints have been set, for example:

Note: breakpoint set at 5 locations.

Such a breakpoint has a single breakpoint entry in the Break/Tracepoint view. Expand the breakpoint entry to see the list of locations where the breakpoint is set. Figure 11-24 shows an example:

Type	Value
● Instr	\INLINE_FN\#11:4
Address	0x000080DC
Address	0x000080EC
Address	0x000080FC
Address	0x0000810C
Address	0x0000811C
Address	0x0000812C
Address	0x0000813C
Address	0x0000814C
Address	0x0000815C
Command	binstr \INLINE_FN\#11:0

Figure 11-24 Breakpoint entry showing multiple locations

Setting hardware breakpoints in inlined functions

If you are setting a hardware breakpoint, then the number of breakpoints that are set is limited by your hardware resources. Hardware breakpoints are set on the locations of the function calls in the order they are found by RealView Debugger.

When insufficient resources are available to set all hardware breakpoint instances, then:

- If no breakpoint resource is available, the following Error message is displayed:
Error V2801C (Vehicle): 0x021a0101: One or more of the requested capabilities could not be set: No HW resource for HW breakpoint.
- If the number of available hardware breakpoint resources is less than the number required for all breakpoint instances, then software breakpoints are set for all instances. In addition, a message is displayed informing you of how many breakpoints have been set, for example:

Note: breakpoint set at 5 locations.

Information: SW breakpoint set - unable to set HW breakpoint

11.15.4 Setting a breakpoint in a C++ template

Setting a breakpoint in a C++ template might result in the breakpoint being set at multiple locations, depending on the breakpoint type.

Setting software breakpoints in a C++ template

If you set a software breakpoint in a C++ template in RAM, then be aware that the breakpoint is set on all locations where the template is instantiated. Software breakpoints are set on the locations of the template instantiations in the order they are found by RealView Debugger. A message is displayed informing you of how many breakpoints have been set, for example:

Note: breakpoint set at 2 locations.

Such a breakpoint has a single breakpoint entry in the Break/Tracepoint view. Expand the breakpoint entry to see the list of locations where the breakpoint is set. Figure 11-25 shows an example:

The screenshot shows the 'Break/Tracepoints' window with one entry. The entry is expanded to show three locations. The first location is an instruction at address 0x0000FDCC, and the second is at address 0x0000FDD8. Both locations have the command 'binstr \TEMPLATE_EX\#20:0'. The window title is 'Break/Tracepoints' and the status bar at the bottom says 'ARM940T_0@RVI'.

Type	Value
● Instr	\TEMPLATE_EX\#20:21
Address	0x0000FDCC
Address	0x0000FDD8
Command	binstr \TEMPLATE_EX\#20:0

Figure 11-25 Breakpoint entry showing multiple C++ template instantiations

Setting hardware breakpoints on a C++ template

If you are setting a hardware breakpoint, then the number of breakpoints that are set is limited by your hardware resources. Hardware breakpoints are set on the locations of the template instantiations in the order they are found by RealView Debugger.

When insufficient resources are available to set all hardware breakpoint instances, then:

- If no breakpoint resource is available, the following Error message is displayed:
Error V2801C (Vehicle): 0x021a0101: One or more of the requested capabilities could not be set: No HW resource for HW breakpoint.
- If the number of available hardware breakpoint resources is less than the number required for all breakpoint instances, then software breakpoints are set for all instances. In addition, a message is displayed informing you of how many breakpoints have been set, for example:

Note: breakpoint set at 2 locations.

Information: SW breakpoint set - unable to set HW breakpoint

11.15.5 Setting a breakpoint at the start of a member function in a C++ class

For a member function of a parent class that is both declared and defined, you can set a breakpoint at the start of that function.

To set a breakpoint at the start of a declared and defined member function:

1. Select **Classes** from the **View** menu to display the Classes view. Figure 11-26 shows an example:

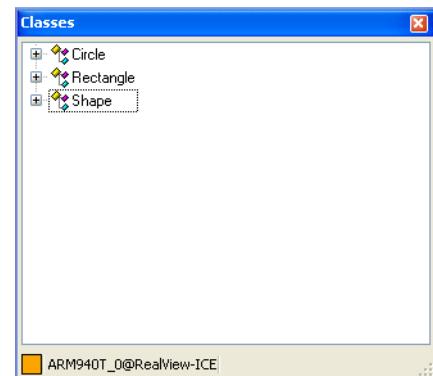


Figure 11-26 Classes view

2. Expand the required class to show the components of that class.
3. Right-click on the required function to display the context menu.
4. Select **Set Break** from the context menu:
 - If the function has a single declaration, a simple breakpoint is set at the start of that function.
 - If the function has multiple declarations, then a List Selection dialog box is displayed showing the declarations. Deselect all declarations except for the one that is of interest, and click **OK**. A simple breakpoint is set at the start of that function.

See also

- *Simple breakpoints* on page 11-6
- *Identifying the components of a class in the Classes view* on page 13-23 for details of the filled stack.

11.15.6 Setting a breakpoint at the location of a Call Stack entry

To set a breakpoint at the location of a Call Stack entry:

1. Select **Call Stack** from the **View** menu to display the Call Stack view, if it is not already visible.
2. Right-click on the Call Stack entry containing the reference where you want to set the breakpoint to display the context menu.
3. Select one of the following options:

Create Breakpoint At...

This displays the Create Breakpoint dialog box. Specify the breakpoint parameters as required, and click **OK** to set the breakpoint. The breakpoint is set at the address defined by the chosen Call Stack entry, if this is permitted.

Create Conditional Breakpoint At...

To display the breakpoint type selection dialog box, shown in Figure 11-27.

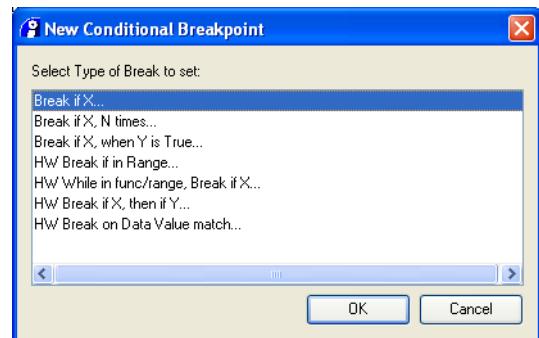


Figure 11-27 Breakpoint type selection dialog box

4. Select the required option from the breakpoint type selection dialog box. The available options depend on the target:
 - **Break if X** displays the Simple Break if X dialog box
 - **Break if X, N Times...** displays the Simple Break if X, N Times dialog box
 - **Break if X, when Y is True...** displays the Simple Break if X, when Y is True dialog box
 - **HW Break if in Range...** displays the HW Break if in Range dialog box
 - **HW While in func/range, Break if X...** displays the HW While in func/range, Break if X dialog box
 - **HW Break if X, then if Y...** displays the HW Break if X, then if Y dialog box.

See also

- *Setting a breakpoint with the Simple Break if X dialog box* on page 11-14
- *Breaking on accesses at multiple locations* on page 11-52
- *Setting breakpoints for location-specific data values* on page 11-54
- *Breaking on a data value match within a range of addresses* on page 11-56
- *Breaking in a function or range that also depends on a data value* on page 11-56
- *Setting a breakpoint that activates after a number of passes* on page 12-13
- *Setting a breakpoint that depends on the result of an expression* on page 12-18.

11.15.7 Setting breakpoints on functions from the Function/Label list

To set a breakpoint on a function using the Function/Label list:

1. Select **Debug → Breakpoints → Set Break/Tracepoint from List → Set from Function/Label list** from the Code window main menu.

The Function Breakpoint Selector dialog box is displayed. Figure 11-28 shows the function list for the example Dhystone image.

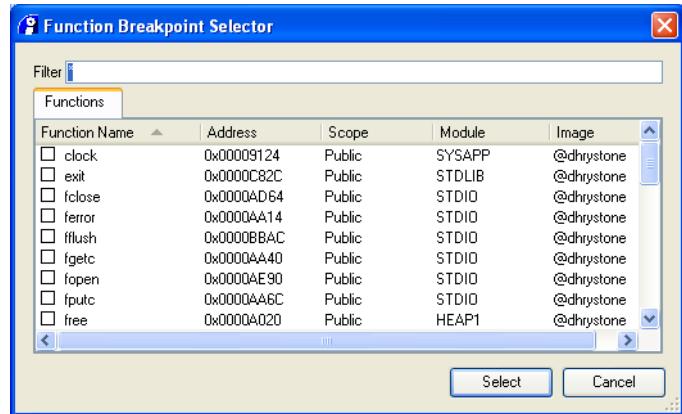


Figure 11-28 Function Breakpoint Selector dialog box

2. For each function where you want to set a breakpoint, select the check box associated with that function.

To narrow the list of functions, enter the characters for the function names that you want to filter. For example, enter **fu*** to show the Func_n functions in the example Dhystone image.

3. Click **Select**. A breakpoint is created at the start of each function that you selected.

11.16 Setting breakpoints for memory accesses

You can set a hardware breakpoint for memory accesses at one or more locations. These breakpoints are called *memory access breakpoints*.

Note

You cannot set breakpoints on accesses to core registers. However, you can set breakpoints on memory mapped registers.

See also:

- *Breakpoint types for memory accesses*
- *Breaking on a named global variable in your source code*
- *Breaking on a variable in the Locals view* on page 11-47
- *Breaking on a variable in the Watch view* on page 11-48
- *Breaking on a Stack address* on page 11-49
- *Breaking on a single memory cell in the Memory view* on page 11-50
- *Breaking on a range of memory cells in the Memory view* on page 11-51
- *Breaking on accesses at multiple locations* on page 11-52
- *Considerations when setting memory access breakpoints* on page 11-53

11.16.1 Breakpoint types for memory accesses

Use the following breakpoint types for memory access breakpoints:

Hardware Access

For breakpoints that activate when a location is read from or written to.

Hardware Read

For breakpoints that activate when a location is read from.

Hardware Write

For breakpoints that activate when a location is written to.

11.16.2 Breaking on a named global variable in your source code

To set a memory access breakpoint on a named global variable:

1. Connect to the target.
2. Load an executable image.

This procedure uses the dhystone image provided in the *RealView Development Suite* (RVDS) examples to demonstrate memory access breakpoints.

3. In the source view, locate the required global variable.
For dhystone, locate `Int_Glob` in `dhry_1.c`.
4. Right-click on the global variable name to display the context menu.
5. Select **Create Breakpoint...** from the context menu. The Create Breakpoint dialog box is displayed.
6. On the Create Breakpoint dialog box, choose the Breakpoint Type you require.
The Value Match field is enabled.
7. Enter **&Int_Glob** in the Location field.

The variable name must be prefixed with &. This indicates that the address of the global variable is to be used. For example, in the dhystone image &Int_Glob corresponds to address 0xE1FC.

8. Leave the Value Match field blank.
9. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box. A breakpoint indicator is placed at the address of the global variable.

See also

- *About setting breakpoints* on page 11-3
- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4.
- *Viewing the target hardware breakpoint support* on page 11-31
- *Breakpoint types for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63
- *Setting breakpoints on custom memory mapped registers* on page 11-67.

11.16.3 Breaking on a variable in the Locals view

To set a memory access breakpoint on a variable in the Locals view:

1. Connect to the target.
2. Load an executable image.

This procedure uses the dhystone image provided in the *RealView Development Suite* (RVDS) examples to demonstrate memory access breakpoints.

3. Run your image, and stop execution at a point of interest.
4. Select **Locals** from the **View** menu of the Code window. The Locals view is displayed, which shows the variables that are currently in scope.
5. Right-click on the global variable name to display the context menu.
6. Select the required breakpoint option from the context menu.

If you select **Create Breakpoint At...** the Create Breakpoint dialog box is displayed, and the address of the chosen variable is entered into the Location field. Do the following:

- a. On the Create Breakpoint dialog box, choose the Breakpoint Type you require. The **Hardware Access** breakpoint type is selected by default.
The Value Match field is enabled.
- b. Leave the Value Match field blank.
- c. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box. A breakpoint indicator is placed at the address of the variable.

If you select **Create Conditional Breakpoint At...** the breakpoint type selection dialog box is displayed. Do the following:

- a. Select the required breakpoint type that you want to set.
- b. Click **OK**. The corresponding dialog box is displayed, and the address of the chosen variable is entered into the Location field.

Note

You cannot set a breakpoint for a variable if <@Rn> appears after the value.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63
- *Setting breakpoints on custom memory mapped registers* on page 11-67
- *Setting a breakpoint that activates after a number of passes* on page 12-13.

11.16.4 Breaking on a variable in the Watch view

To set a memory access breakpoint on a variable in the Watch view:

1. Connect to the target.
2. Load an executable image.
3. Run your image, and stop execution at a point of interest.
4. Select **Watch** from the **View** menu of the Code window. The Watch view is displayed.
5. Right-click on the variable name to display the context menu.
6. Select the required breakpoint option from the context menu.

If you select **Create Breakpoint At...** the Create Breakpoint dialog box is displayed, and the address of the chosen variable is entered into the Location field. Do the following:

- a. On the Create Breakpoint dialog box, choose the Breakpoint Type you require. The **Hardware Access** breakpoint type is selected by default.
The Value Match field is enabled.
- b. Leave the Value Match field blank.
- c. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box. A breakpoint indicator is placed at the address of the variable.

If you select **Create Conditional Breakpoint At...** the breakpoint type selection dialog box is displayed. Do the following:

- a. Select the required breakpoint type that you want to set.
- b. Click **OK**. The corresponding dialog box is displayed, and the address of the chosen variable is entered into the location field.

Note

You cannot set a breakpoint for a variable if <@Rn> appears after the value.

7. On the Create Breakpoint dialog box, choose the Breakpoint Type you require. The **Hardware Access** breakpoint type is selected by default.
The Value Match field is enabled.
8. Leave the Value Match field blank.

9. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box. A breakpoint indicator is placed at the address of the global variable.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63
- *Setting breakpoints on custom memory mapped registers* on page 11-67
- *Setting a breakpoint that activates after a number of passes* on page 12-13.

11.16.5 Breaking on a Stack address

To set a memory access breakpoint at a Stack address:

1. Connect to the target.
2. Load an executable image.
3. Select **Stack** from the **View** menu to display the Stack view. Figure 11-29 shows an example:

Stack	
0x07FFFF70	SP 0xE7FF0010
0x07FFFF74	0xE800E800
0x07FFFF78	0x00000002
0x07FFFF7C	0x0000C4E4
0x07FFFF80	0x00010F50
0x07FFFF84	0x00000000
0x07FFFF88	0x00000000
0x07FFFF8C	0x00000000
0x07FFFF90	0x00010F8C
ARM7TDMI@RVISS	

Figure 11-29 Example Stack view

4. Right-click on the contents of the required stack entry to display the context menu.
 5. Select **Create Breakpoint At...** from the context menu.
- The Create Breakpoint dialog box is displayed. The Stack address is inserted in the Location field, and the breakpoint type is set to **Hardware Access** by default. Also, the Value Match field is enabled.
6. Change the breakpoint type, if required.
 7. Leave the Value Match field blank.
 8. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Breakpoint types for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63
- *Setting breakpoints on custom memory mapped registers* on page 11-67

- *Viewing the Stack* on page 13-58
- *Changing the stack pointer* on page 14-21.

11.16.6 Breaking on a single memory cell in the Memory view

To set a memory access breakpoint on a single memory cell in the Memory view:

1. Connect to the target.
2. Load an executable image.
3. Select **Memory** from the **View** menu to display the Memory view.
4. Set the Start address to the start of the memory area of interest. Figure 11-30 shows an example:

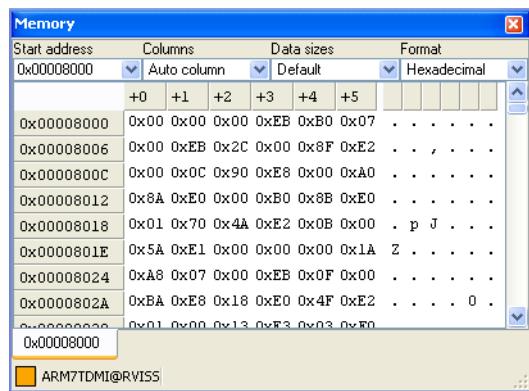


Figure 11-30 Example Memory view

5. Set the Data sizes field in the toolbar to the required size of the memory cells:
 - Select **Default** or **1 byte** if you want to set a breakpoint at a single address memory cell.
 - Select any other size if you want to set a breakpoint on a multi-address memory cell. For example, select **4 byte** to set a breakpoint on a four-byte memory cell.
 6. Right-click on the contents of the required memory location to display the context menu.
 7. Select **Create Breakpoint At...** from the context menu. The Create Breakpoint dialog box is displayed:
 - For a single address memory cell, the address is inserted in the Location field.
 - For a multi-address memory cell, the address range occupied by that memory cell is inserted in the Location field.
- Also, the breakpoint type is set to **Hardware Access** by default and the Value Match field is enabled.
8. Change the breakpoint type, if required.
 9. Leave the Value Match field blank.
 10. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box.

See also

- *Viewing the target hardware breakpoint support* on page 11-31

- *Breakpoint types for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63
- *Setting breakpoints on custom memory mapped registers* on page 11-67.

11.16.7 Breaking on a range of memory cells in the Memory view

To set a memory access breakpoint on a range of memory cells in the Memory view:

1. Connect to the target.
2. Load an executable image.
3. Select **Memory** from the **View** menu to display the Memory view.
4. Set the Start address to the start of the memory area of interest.
5. Select the range of memory cells for which the breakpoint is to be set. Figure 11-31 shows an example:

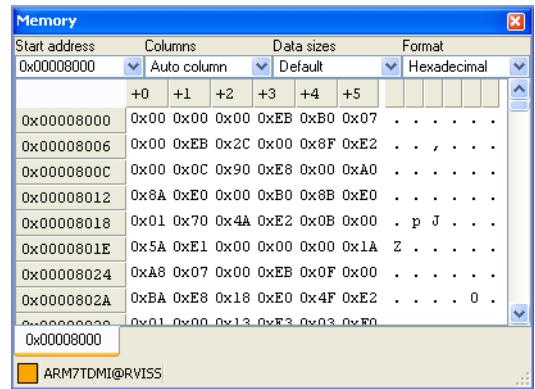


Figure 11-31 Example Memory view showing selected locations

6. Right-click on one of the selected memory locations to display the context menu.
7. Select **Create Breakpoint At...** from the context menu to display the Create Breakpoint dialog box. Figure 11-32 shows an example:

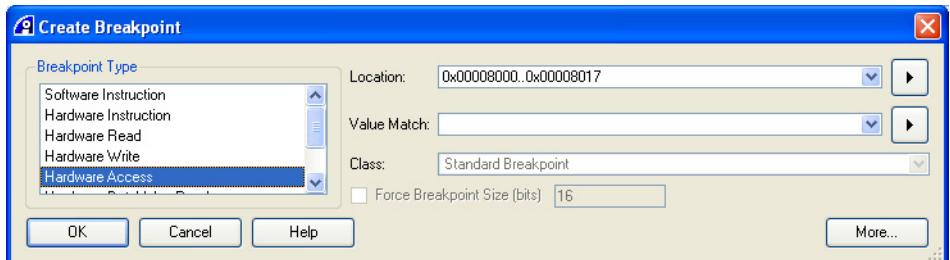


Figure 11-32 Create Breakpoint dialog box showing selected address range

The selected address range is inserted in the Location field, and the breakpoint type is set to **Hardware Access** by default. Also, the Value Match field is enabled.

8. Change the breakpoint type, if required.

9. Leave the Value Match field blank.
10. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Breakpoint types for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Setting breakpoints for location-independent data values* on page 11-59
- *Chaining hardware breakpoints* on page 11-63
- *Setting breakpoints on custom memory mapped registers* on page 11-67.

11.16.8 Breaking on accesses at multiple locations

To set memory access breakpoints on accesses at multiple locations:

1. Connect to the target.
2. Load an executable image.

This procedure uses the dhystone image provided in the RVDS examples to demonstrate memory access breakpoints.

3. Select **Debug → Breakpoints → Hardware → HW Break if X, then if Y...** from the Code window main menu to display the HW Break if X, then if Y dialog box.

Figure 11-33 shows an example:

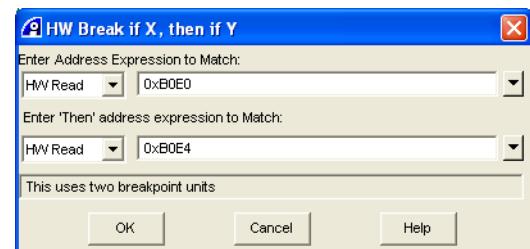


Figure 11-33 HW break if X, then if Y dialog box

4. Choose the type of hardware breakpoint that you want to set for the first breakpoint unit. The default is **HW Read**.
5. Specify the location for the first breakpoint unit (X), which can be:
 - a specific address, which can be the address of a variable or function, or an address returned from a macro
 - an address range
 - a function.
6. Choose the type of hardware breakpoint that you want to set for the second breakpoint unit. The default is **HW Read**.
7. Specify the location for the second breakpoint unit (Y), which can be:
 - a specific address, which can be the address of a variable or function, or an address returned from a macro
 - an address range
 - a function.
8. Click **OK** to set the breakpoint units, and close the dialog box.

This creates a breakpoint chain. Figure 11-34 shows an example. Chained breakpoints activate only when all breakpoint units have activated.

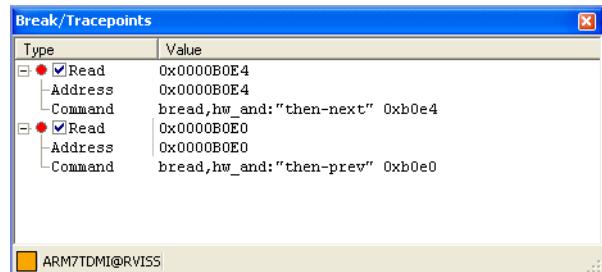


Figure 11-34 Chained breakpoints

See also

- [Specifying address ranges](#) on page 11-10
- [Viewing the target hardware breakpoint support](#) on page 11-31
- [Setting breakpoints on functions](#) on page 11-39
- [Setting breakpoints for location-specific data values](#) on page 11-54
- [Setting breakpoints for location-independent data values](#) on page 11-59
- [Chaining hardware breakpoints](#) on page 11-63
- [Setting breakpoints on custom memory mapped registers](#) on page 11-67.

11.16.9 Considerations when setting memory access breakpoints

When you set a memory access breakpoint:

- RealView Debugger sets a breakpoint on a symbol address where it exists on the stack. As soon as you exit the function, the address is no longer meaningful. Therefore, do not use such a breakpoint where execution runs past the function return call.
- Unlike the Watch view, the Stack view acts like a snapshot of a chosen address. It does not track each invocation of a function and so is not able to track the chosen symbol.

11.17 Setting breakpoints for location-specific data values

You can set a memory access breakpoint for accesses to a specific data value at a specific location.

Note

You cannot set breakpoints on core registers. However, you can set breakpoints on memory mapped registers.

See also:

- *Breakpoint access types for accesses to location-specific data values*
- *Breaking on the value of a global variable*
- *Breaking on the value of a watched variable* on page 11-55
- *Breaking on a data value match within a range of addresses* on page 11-56
- *Breaking in a function or range that also depends on a data value* on page 11-56
- *Setting breakpoints on custom memory mapped registers* on page 11-67.

11.17.1 Breakpoint access types for accesses to location-specific data values

Use the following breakpoint access types for accesses to location-specific data values:

Hardware Access

For breakpoints that activate when a specific value is read from or written to a location.

Hardware Read

For breakpoints that activate when a specific value is read from a location.

Hardware Write

For breakpoints that activate when a specific value is written to a location.

11.17.2 Breaking on the value of a global variable

To set a memory access breakpoint on the value of a global variable:

1. Connect to the target.
2. Load an executable image.

This procedure uses the dhystone image provided in the RVDS examples.

3. Select **Symbols** from the **View** menu to display the Symbols view.
4. Click the **Variables** tab to display the list of variables.
5. Right-click on the global variable name to display the context menu.
For dhystone, right-click on the Int_Glob variable.

6. Select **Create Breakpoint...** to display the Create Breakpoint dialog box.

The variable name is inserted in the Location field, and is prefixed with &. This indicates that the address of the global variable is to be used. For example, for the dhystone image &@dhystone\\Int_Glob corresponds to address 0xEA8.

7. On the Create Breakpoint dialog box, choose the Breakpoint Type you require. The default is type is **Hardware Access**.

The Value Match field is enabled.

8. Enter the required value in the Value Match field.
For dhystone, enter **5**.
9. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box. A breakpoint indicator is placed at the address of the global variable.

See also

- *About setting breakpoints* on page 11-3
- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- *Help with specifying locations and values for hardware breakpoints* on page 11-11
- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints for memory accesses* on page 11-46
- *Breakpoint access types for accesses to location-specific data values* on page 11-54
- *Setting breakpoints for location-specific data values* on page 11-54
- *Chaining hardware breakpoints* on page 11-63.

11.17.3 Breaking on the value of a watched variable

To set a memory access breakpoint on the value of a watched variable:

1. Connect to the target.
2. Load an executable image.
This procedure uses the dhystone image provided in the RVDS examples.
3. Set a watch variable.
4. Right-click on the watch variable name in the Watch view to display the context menu.

————— Note —————

If the watched variable is an array, first expand the array in the Watch view if you want to set a breakpoint for an element of the array.

5. Select the required breakpoint option from the context menu.

If you select **Create Breakpoint At...** the Create Breakpoint dialog box is displayed, and the address of the chosen variable is entered into the Location field. Do the following:

- a. Choose the Breakpoint Type you require.
The Value Match field is enabled.
- b. Enter the required value in the Value Match field.
- c. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box. A breakpoint indicator is placed at the address of the watched variable.

If you select **Create Conditional Breakpoint At...** the breakpoint type selection dialog box is displayed. Do the following:

- a. Select **HW Break on Data Value match...**
- b. Click **OK**. The HW Break on Data Value match dialog box is displayed, and the address of the chosen variable is entered into the location field.
- c. Enter the required value in the field at the bottom of the dialog box.
- d. Click **OK** to set the breakpoint and close the HW Break on Data Value match dialog box. A breakpoint indicator is placed at the address of the watched variable.

Note

You cannot set a breakpoint for a variable if <@Rn> appears after the value.

See also

- *Help with specifying locations and values for hardware breakpoints* on page 11-11
- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints for memory accesses* on page 11-46
- *Breakpoint access types for accesses to location-specific data values* on page 11-54
- *Setting breakpoints for location-specific data values* on page 11-54
- *Chaining hardware breakpoints* on page 11-63
- *Setting a breakpoint that activates after a number of passes* on page 12-13
- *Setting watches* on page 13-67.

11.17.4 Breaking on a data value match within a range of addresses

To set a memory access breakpoint on a data value match within a range of addresses:

1. Connect to the target.
2. Load an executable image.
- This procedure uses the dhrystone image provided in the RVDS examples.
3. Select **Debug → Breakpoints → Hardware → HW Break if in Range...** from the Code window main menu. The HW Break if in Range dialog box is displayed. Figure 11-35 shows an example:



Figure 11-35 HW Break if in Range dialog box

4. Choose the type of hardware breakpoint that you want to set. The default is **HW Access**.
5. Specify the range of data addresses.
For example, if you are using the dhrystone image, enter `0x8DA8..0x8EB7`.
6. If you also want to match on a specific data value:
 - a. Select the **And if Data Value matches** check box.
 - b. Enter the required data value in the data field.
7. Click **OK** to set the breakpoint and close the dialog box.

11.17.5 Breaking in a function or range that also depends on a data value

To set a breakpoint in a function or range that also depends on a data value:

1. Connect to the target.
2. Load an executable image.

This procedure uses the dhystone image provided in the RVDS examples.

3. Select **Debug → Breakpoints → Hardware → HW While in function/range, Break if X...** from the Code window main menu. The HW While in function/range, Break if X dialog box is displayed. Figure 11-36 shows an example:

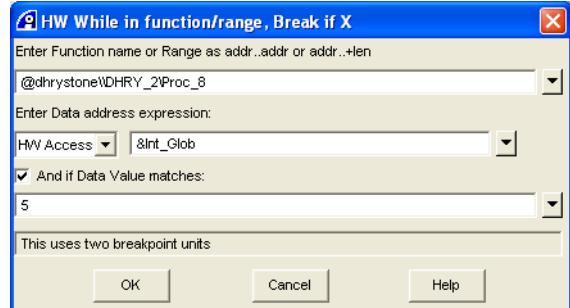


Figure 11-36 HW While in func/range, Break if X dialog box

4. Specify the location for the first breakpoint unit, which can be:
 - an address range
 - a function.
 For example:
`@dhystone\\DHRY_2\\Proc_8`
5. For the second breakpoint unit, choose the type of hardware breakpoint that you want to set. The default is **HW Access**.
6. Enter the data address expression that must be accessed to activate the breakpoint. For example, enter **&Int_Glob**.
7. If you also want to match on a specific data value:
 - a. Select the **And if Data Value matches** check box.
 - b. Enter the required data value in the data field.
8. Click **OK** to set up the breakpoint units and close the dialog box.

This creates a breakpoint chain. Figure 11-37 shows an example. Chained breakpoints activate only when all breakpoint units have activated.

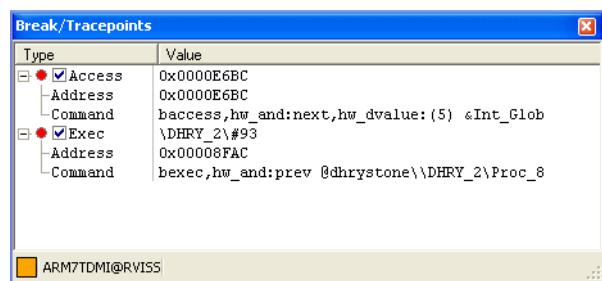


Figure 11-37 Chained breakpoint

See also

- *Specifying address ranges* on page 11-10
- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints on functions* on page 11-39

- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Chaining hardware breakpoints* on page 11-63.

11.18 Setting breakpoints for location-independent data values

You can set a memory access breakpoint for accesses to a specific data value, regardless of the location. These breakpoints are also called *data only breakpoints*.

See also:

- *Breakpoint access types for data only breakpoints*
- *Setting a data only breakpoint*
- *Breaking on a data value match with a modifier* on page 11-60
- *Considerations when setting data only breakpoints* on page 11-61.

11.18.1 Breakpoint access types for data only breakpoints

Use the following access types for data only breakpoints:

Hardware DataValue Access

For breakpoints that activate when a specific data value is read from or written to any location.

Hardware DataValue Read

For breakpoints that activate when a specific data value is read from any location.

Hardware DataValue Write

For breakpoints that activate when a specific data value is written to any location.

11.18.2 Setting a data only breakpoint

To set a data only breakpoint:

1. Connect to the target.
2. Load an executable image.

This procedure uses the dhrystone image provided in the RVDS examples.

3. Select **Debug** → **Breakpoints** → **Create Breakpoint...** from the Code window main menu to display the Create Breakpoint dialog box.

4. On the Create Breakpoint dialog box, choose the Breakpoint Type you require.

For example, select **Hardware DataValue Access**.

The Location field is disabled, and the Value Match field is enabled.

5. Enter the required value in the Value Match field.

6. Click **OK** to set the breakpoint and close the Create Breakpoint dialog box.

In this example, the breakpoint activates when the specified value is read from or written to any address.

7. If you want to see the breakpoint details, then display the Break/Tracepoints view.

See also

- *Help with specifying locations and values for hardware breakpoints* on page 11-11
- *Viewing the breakpoint details* on page 11-21
- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54

- *Breakpoint access types for data only breakpoints* on page 11-59
- *Chaining hardware breakpoints* on page 11-63.

11.18.3 Breaking on a data value match with a modifier

To set a data only breakpoint for a data value match with a modifier:

1. Connect to the target.

2. Load an executable image.

This procedure uses the dhrystone image provided in the RVDS examples.

3. Select **Debug → Breakpoints → Hardware → HW Break on Data Value match...** from the Code window main menu. The HW Break on Data Value match dialog box is displayed. Figure 11-38 shows an example:

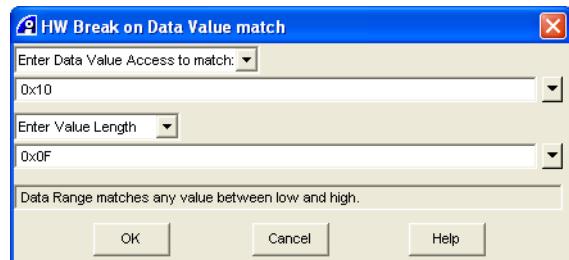


Figure 11-38 HW Break on Data Value match dialog box

4. Choose the type of hardware data value breakpoint that you want to set for the match:
 - **Enter Data Value Access to Match**
 - **Enter Data Value Read to Match**
 - **Enter Data Value Write to Match.**
5. Enter the required data value in the data field.
6. Choose a data value modifier, if your hardware supports it:
Enter None (no range)

Match the first data value only. This is the default.

Enter End Value

Match values in the range from the first data value and the second data value specified, inclusive.

Enter Value Length

Match values in the range from the first data value to the first data value plus the length specified, inclusive.

For example, if the first data value is 0x10, and the length is 0xF, then the range is from 0x10 to 0x1F, inclusive.

Enter Value Mask

Specify a value mask if you are interested only in certain bits of the value. For example, if you have a data field occupying five bits (0x1F), then set a mask of 0x5 if only bits 0 and 2 are of interest.

7. Enter the required value.
8. Click **OK** to set the breakpoint and close the dialog box.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Chaining hardware breakpoints* on page 11-63.

11.18.4 Considerations when setting data only breakpoints

When you set a data only breakpoint:

- you can set only a single breakpoint of each type
- no breakpoint icon is displayed in the code view for data only breakpoints.

11.19 Forcing the size of a software breakpoint

The default size of a software breakpoint is 32 bits if there is no image information. If there is image information, then that determines the size of the breakpoint. However, you can change this if required.

To force the size of a software breakpoint:

1. Either:
 - create a new software breakpoint, and leave the Create Breakpoint dialog box open
 - edit an existing software breakpoint to display the Edit Breakpoint dialog box.
2. Select the **Force Breakpoint Size (bits)** check box.
3. Enter the required size:
 - enter **16** for 16-bit (Thumb code)
 - enter **32** for 32-bit (ARM code).

————— **Note** —————

Also, use Thumb breakpoints for Thumb2 and Thumb2EE.

4. Click **OK** to set the breakpoint.

See also:

- *Editing a breakpoint* on page 11-25.

11.20 Chaining hardware breakpoints

Where supported by your target hardware, you can combine hardware breakpoints to create complex tests. This is called *chaining*. Each individual breakpoint used in the chain is called a *breakpoint unit*. You normally use CLI command qualifiers to chain breakpoints together.

See also:

- *CLI command qualifiers used for chaining breakpoints*
- *Converting existing individual hardware breakpoints to chained breakpoints*
- *Considerations when chaining hardware breakpoints* on page 11-64.

11.20.1 CLI command qualifiers used for chaining breakpoints

You use the `hw_and` command qualifier to create chained breakpoints:

- For the first breakpoint unit in the chain, include the `hw_and:next` or `hw_and:"then-next"` command qualifiers. You must create this breakpoint unit before any other breakpoint unit in the chain.
 - For each subsequent breakpoint unit in the chain, include the `hw_and:prev` or `hw_and:"then-prev"` command qualifiers.
- Alternatively, use the `hw_and:id` or `hw_and:"then-id"` command qualifiers.

If you have created individual breakpoints, and you want to chain them together, you must use CLI commands to create new breakpoints based on the existing breakpoints. You must create new breakpoints because you cannot use the `modify` qualifier with the `hw_and` qualifier. However, you can modify an existing chained breakpoint with any other qualifier and also change the address expression.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Breaking in a function or range that also depends on a data value* on page 11-56
- *Breaking on accesses at multiple locations* on page 11-52
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.20.2 Converting existing individual hardware breakpoints to chained breakpoints

To create chained breakpoints based on existing hardware breakpoints:

1. Display the Break/Tracepoints view.
2. Expand the breakpoint instances to determine the CLI command that was used to create each breakpoint.
3. Make a note of the commands used to create each breakpoint. Alternatively:
 - copy the commands from the output in the **Cmd** tab, and save them in a text file
 - copy the commands from the log or journal file, if you have one.
4. Clear the original breakpoints that you are using to create the chained breakpoints.
5. Determine the order in which the breakpoints are to be chained.

6. For each breakpoint command you enter, include the `hw_and` qualifier to set up the chained breakpoints as follows:
 - a. For the first breakpoint in the chain, include the `hw_and:next` or `hw_and:"then-next"` command qualifiers.
 - b. For each subsequent breakpoint in the chain, include the `hw_and:prev` or `hw_and:"then-prev"` command qualifiers.

Alternatively, use the `hw_and: id` or `hw_and:"then- id"` command qualifiers.

See also

- *Viewing the target hardware breakpoint support* on page 11-31
- *Breaking in a function or range that also depends on a data value* on page 11-56
- *Breaking on accesses at multiple locations* on page 11-52
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.20.3 Considerations when chaining hardware breakpoints

Be aware of the following when setting chained breakpoints:

- a breakpoint chain activates only when all breakpoint units have activated
- only hardware breakpoints can be chained together
- you cannot edit a breakpoint that is a member of a breakpoint chain.

11.21 Specifying processor exceptions (global breakpoints)

Where supported by your debug target, RealView Debugger maintains a list of processor exceptions that automatically activate a breakpoint in any application program. These are global breakpoints, that is they apply to processor exceptions and not addresses. During your debugging session you can examine this list and select, or deselect, events that halt the processor.

— Note —

The available processor exceptions depend on your target.

To specify the processor exceptions that automatically stop execution:

1. Select **Processor Exceptions...** from the **Debug** menu to display the Processor Exceptions dialog box. Figure 11-39 shows an example, which lists the processor exceptions for an ARM architecture-based processor connected through RealView ICE.

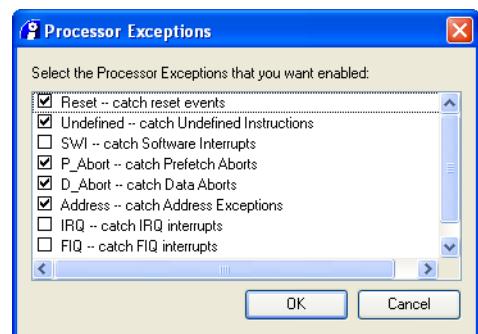


Figure 11-39 Processor Exceptions list selection dialog box

2. To modify the processor exceptions that automatically stop execution:
 - a. Select the check box of each exception that you want to change.
 - b. Click **OK** to make the selected exceptions active.

See also:

- *Specifying a processor exception that runs a macro when triggered*
- *Considerations when setting SVC vector catch (hardware targets)* on page 11-66
- *Attaching a macro to a global breakpoint* on page 12-22
- *Viewing semihosting controls for DSTREAM or RealView ICE JTAG connections* on page 13-35
- *Viewing semihosting controls for RVISS targets* on page 13-37.

11.21.1 Specifying a processor exception that runs a macro when triggered

To specify a processor exception that runs a macro when it is triggered, use the **BGLOBAL** command. The macro return value determines whether execution continues or stops.

— Note —

If no macro is specified, execution always stops when selected exception is triggered.

For example, to run the macro `my_macro()` when the Prefetch Abort exception is triggered on an ARM architecture-based processor connected through DSTREAM or RealView ICE, enter:

1. Load the macro definition.

2. Specify the BGLOBAL command:

```
bglobal,enable "prefetch abort" ; my_macro()
```

See also

- Chapter 15 *Debugging with Command Scripts*
- Chapter 16 *Using Macros for Debugging*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.21.2 Considerations when setting SVC vector catch (hardware targets)

For connections to non ARMv7-M hardware processors, semihosting uses the SVC vector by default. For these processors, you must not enable the SVC vector catch with semihosting enabled:

- If you enable both in the connection properties, the following warning is displayed when you connect:
Warning: ARM_config: cannot have vector catch of SVC and semi-hosting enabled.
- If you attempt to enable the SVC vector catch using the BGLOBAL command or the Processor Exceptions dialog box with semihosting enabled, the following error is displayed:
Error V2801C (Vehicle): vector-catch-svc not available. This resource may be in use for semihosting.

The SVC vector catch is not enabled in this case.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring vector catch* on page 3-24
 - *Configuring Semihosting* on page 3-29.
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the BGLOBAL command.

11.22 Setting breakpoints on custom memory mapped registers

If you have created custom registers that are mapped to specific memory locations for a connection, then you can set hardware data access breakpoints on those registers.

————— Note —————

Memory mapping does not have to be enabled for you to have access to the custom memory mapped registers.

See also:

- *Breaking on a custom memory mapped register*
- *Breaking on a data value of a custom memory mapped register* on page 11-68

11.22.1 Breaking on a custom memory mapped register

To set a breakpoint that tests when a custom register is accessed:

1. Set up the custom registers for your target connection.

You can do this by assigning BCD files that contain custom registers to your connection (such as the CP.bcd file provided with RealView Debugger).

————— Note —————

Setting up custom registers is outside the scope of this document.

2. Connect to the target.

3. Load your image.

4. Select **Debug → Breakpoints → Conditional → Break if X...** from the Code windows main menu to display the Simple Break if X dialog box. Figure 11-40 shows an example:

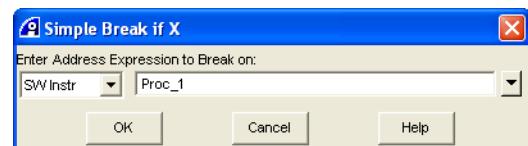


Figure 11-40 Simple Break if X dialog box

If you have previously set a breakpoint using this dialog box, then the details of the last breakpoint you set are shown.

5. Select the breakpoint type.

For custom registers, select one of the following:

HW Access

For breakpoints that activate when the register location is read from or written to.

HW Read

For breakpoints that activate when the register location is read from.

HW Write

For breakpoints that activate when the register location is written to.



6. Click the expression selector button, to select from:

- various lists, including your Favorites List
- expressions used in previous breakpoints during this session.

7. Select <Register list...> from the menu to display the Register List Selection dialog box. Figure 11-41 shows an example:



Figure 11-41 Register List Selection dialog box

8. Select the custom register from the list.
9. Click **OK** to close the Register List Selection dialog box. The register expression is inserted into the Simple Break if X dialog box.
The register name is preceded by `reg@`, for example, `reg:@G_SC_PCI`.
10. Click **OK** to set the breakpoint and close the Simple Break if X dialog box.

See also

- *Connecting to a target* on page 3-27.
- *Loading an executable image* on page 4-4
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

11.22.2 Breaking on a data value of a custom memory mapped register

To set a breakpoint on a data value of a custom memory mapped register:

1. Set up the custom registers for your target connection.
You can do this by assigning BCD files that contain custom registers to your connection (such as the CP.bcd file provided with RealView Debugger).

————— Note —————

Setting up custom registers is outside the scope of this document.

2. Connect to the target.
3. Load your image.
4. Set a breakpoint on the required custom register.
5. Display the Break/Tracepoints view, if it is not already visible.
6. In the Break/Tracepoints view, right-click on the breakpoint that you have set on the custom register to display the context menu.
7. Select **Edit Breakpoint** from the context menu to display the Edit Breakpoint dialog box.
8. Enter the required value in the Value Match field.

9. Click **OK** to set the breakpoint and close the dialog box.

See also

- *Connecting to a target* on page 3-27.
- *Loading an executable image* on page 4-4
- *Help with specifying locations and values for hardware breakpoints* on page 11-11.
- *Breaking on a custom memory mapped register* on page 11-67
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

11.23 Setting breakpoints from the breakpoint history list

Breakpoints are added to a history list if you set them:

- directly in the source or disassembly view
- using the breakpoint dialog boxes, for example the Create Breakpoint or the Simple Break if X dialog box.

Note

If you set a breakpoint in another way, for example using a CLI command, this is not added to the history list.

This is useful if you have previously set up a complex breakpoint (with conditions and actions), and you want to re-use that breakpoint, or use it as the basis for a new breakpoint.

The breakpoint history list is stored in your personal history file, `exphist.sav`, which is saved in your RealView Debugger home directory. Your personal history file is updated when you exit RealView Debugger at the end of your debugging session, and it contains a snapshot of the current breakpoints across all your debug targets. The items in the breakpoint history list include breakpoints you have set during the current debugging session, and breakpoints you have set from previous debugging sessions.

Note

If you are using RealView Debugger on non-Windows platforms, the history file is created only if you create and save a favorite, for example a breakpoint or watchpoint.

To set a breakpoint from the breakpoint history list:

1. Select **Debug → Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint History...** from the Code window main menu to display the breakpoint history list dialog box. Figure 11-42 shows an example:

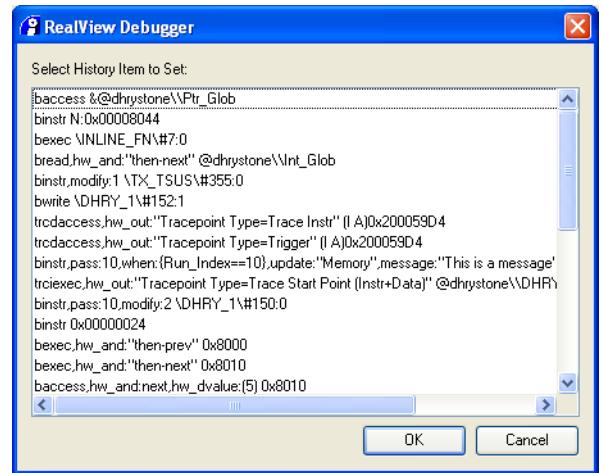


Figure 11-42 Breakpoint history list dialog box

2. Select the breakpoint that you want to set.
3. Click **OK** to set the breakpoint and close the dialog box.

The breakpoint is set.

Note

If you want to use this breakpoint as the basis for a new breakpoint, then edit the breakpoint.

See also:

- *Unconditional and conditional breakpoints* on page 11-6
- *Editing a breakpoint* on page 11-25
- *Creating new breakpoint favorites* on page 11-72
- Appendix E *RealView Debugger on Red Hat Linux*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.24 Creating new breakpoint favorites

You can create new breakpoint favorites directly. However, you must be familiar with the CLI command syntax for the breakpoint commands.

To create a new breakpoint favorite:

1. Connect to the target.

————— **Note** —————

You must connect to a target before you can use or modify your Favorites List.

2. Load your image.
3. Familiarize yourself with the CLI command syntax for the breakpoint commands.
4. Select **Debug → Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...** from the Code window main menu to display the Favorites/Chooser Editor dialog box. Figure 11-43 shows an example. Any previous breakpoints that have been added to the list are shown.

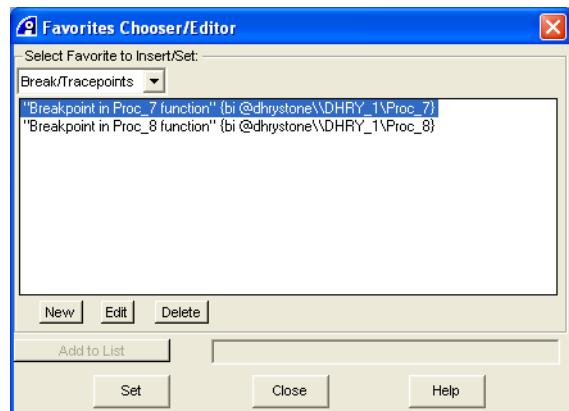


Figure 11-43 Favorites/Chooser Editor dialog box

5. Click **New** to display the New/Edit Favorite dialog box, shown in Figure 11-44.

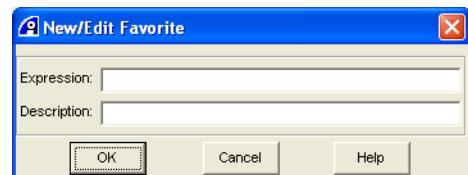


Figure 11-44 New/Edit Favorite dialog box

6. Enter the CLI command for the breakpoint in the Expression field.
7. Optionally, enter a short text description to identify the breakpoint for future use.
8. Click **OK** to confirm the entries and close the New/Edit Favorite dialog box.

The Favorites Chooser/Editor dialog box is displayed with the newly-created breakpoint shown in the display list.

9. Either:
 - Click **Set** if you want to set the breakpoint.
 - Click **Cancel** if you do not want to set the breakpoint at this time.

The Favorites/Chooser Editor dialog box closes.

See also:

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

11.25 Setting breakpoints from your Favorites List

When you first start to use RealView Debugger, your personal Favorites List is empty. You can create breakpoints and add them directly to this list or you can add breakpoints that you have been using in the current debugging session.

RealView Debugger keeps a record of all breakpoints that you set during your debugging session as part of your history file. By default, at the end of your debugging session, these processor-specific lists are saved in the file `exphist.sav` in your RealView Debugger home directory. This file also keeps a record of your favorites, for example Break Qualifiers, Break Actions, and Break/Tracepoints.

Note

You must connect to a target before you can use or modify your Favorites List.

To set a breakpoint from your favorites list:

1. Select **Debug → Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...** from the Code window main menu to display the Favorites/Chooser Editor dialog box. Figure 11-45 shows an example. Any previous breakpoints that have been added to the list are shown.

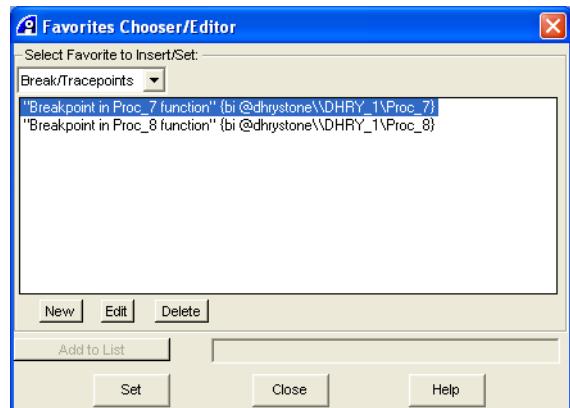


Figure 11-45 Favorites/Chooser Editor dialog box

2. Select the breakpoint that you want to set from your favorites list.
3. Click **Set** to set the breakpoint and close the dialog box.

Chapter 12

Controlling the Behavior of Breakpoints

This chapter explains how you can modify the behavior of breakpoints so that the activation of a breakpoint can be delayed and, when activated, what actions it performs. It includes:

- *About controlling the behavior of breakpoints* on page 12-2
- *Updating windows and views when a breakpoint activates* on page 12-4
- *Displaying user-defined messages when a breakpoint activates* on page 12-7
- *Setting the execution behavior for a breakpoint* on page 12-9
- *Setting breakpoints that test for hardware input triggers* on page 12-10
- *Setting a breakpoint that activates after a number of passes* on page 12-13
- *Resetting breakpoint pass counters* on page 12-17
- *Setting a breakpoint that depends on the result of an expression* on page 12-18
- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- *Setting a breakpoint on a specific instance of a C++ class* on page 12-24
- *Example of breakpoint behavior* on page 12-27.

12.1 About controlling the behavior of breakpoints

By default, a breakpoint activates when it is hit, and then always causes execution to stop. However, you might want to control:

- when the breakpoint activates
- what happens after it activates.

See also:

- *Features for controlling breakpoint behavior*
- *Considerations when using software conditions*
- *Conditional hardware breakpoints* on page 12-3.

12.1.1 Features for controlling breakpoint behavior

You can control the behavior of a breakpoint using the following features:

- Software conditions, which enable you to specify when a breakpoint activates. Breakpoints that have software conditions assigned to them are called conditional breakpoints. These breakpoint conditions are controlled by RealView® Debugger.
- Breakpoint actions, which enable you to specify what happens after a breakpoint has activated.

See also

- *Events that determine when a breakpoint is hit* on page 1-36
- *Actions that can be performed when a breakpoint activates* on page 1-36
- *Conditional breakpoint activation* on page 1-37
- *Unconditional and conditional breakpoints* on page 11-6.

12.1.2 Considerations when using software conditions

Using software conditions can be very intrusive because RealView Debugger takes control when the breakpoint activates. The specified conditions are checked and then, if applicable, control is returned to the application.

When a conditional breakpoint activates, there might be a discrepancy between the real state of your target and what is shown in the Code window. For example, the State field might show that the target has stopped when it is running. This is because the state of the target, as reported by RealView Debugger, is a snapshot of the target status at the time that the breakpoint is activated. Depending on the tasks being performed on your host system, the target state might have changed. Therefore, when you are using conditional breakpoints, remember that the state of the target depends on several factors, including:

- the clock speed of your development platform and the targets it contains
- those instructions currently being executed on the target
- how much processing is required on the host to resolve the software conditions.

Note

If you are debugging OS-aware applications in RSD mode, you can set OS-aware breakpoints.

See also

- *Example of breakpoint behavior* on page 12-27

- the following in the *RealView Debugger RTOS Guide*:
 - Chapter 7 *Debugging Your OS Application*.

12.1.3 Conditional hardware breakpoints

Although hardware breakpoints can have conditions that are controlled in hardware, they are not conditional breakpoints in RealView Debugger. RealView Debugger recognizes a breakpoint as conditional only when a software condition is assigned to the breakpoint.

If you have created a hardware breakpoint without any software conditions assigned, then you must edit the breakpoint to assign one or more software conditions to it.

See also

- *Editing a breakpoint* on page 11-25.

12.2 Updating windows and views when a breakpoint activates

You can update any or all of the RealView Debugger windows and views when a breakpoint activates.

See also:

- *Updating specific windows and views*
- *Updating all windows and views* on page 12-6.

12.2.1 Updating specific windows and views

To update specific windows and views when a breakpoint activates:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed, shown in Figure 12-1.

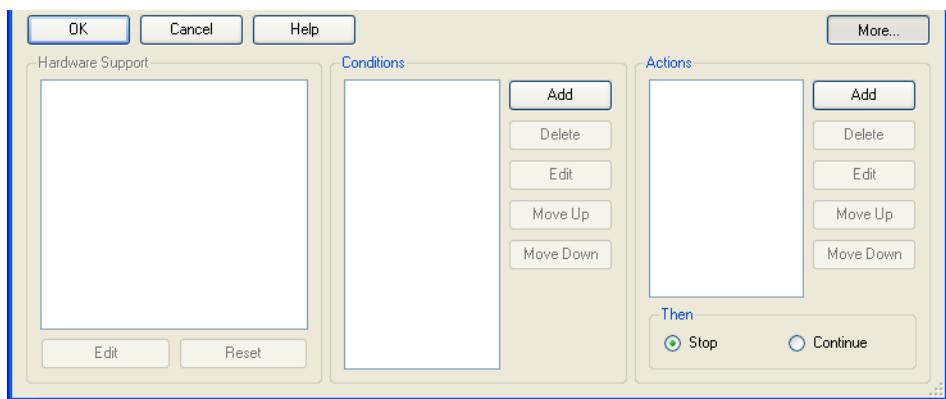


Figure 12-1 Breakpoint behavior controls

3. Click **Add** in the Actions group. The Add Action dialog box is displayed, shown in Figure 12-2.

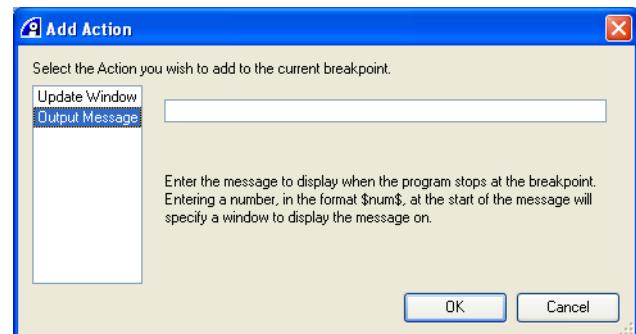


Figure 12-2 Add Action dialog box

4. Select the **Update Window** action. This is the default.
5. Select the window or view that you want to update when the breakpoint activates. For example, select **Watch** to update the Watch view.

————— Note —————

You can also update any user-defined windows that you have opened.

- Click **OK** to close the Add Action dialog box. The action is added to the breakpoint action list, shown in Figure 12-3.

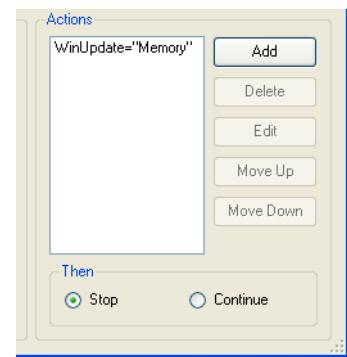


Figure 12-3 The breakpoint action list

- If you want more windows or views to be updated, repeat this operation for each window or view.
- When you have finished adding the windows and views to be updated, click **OK** to set the breakpoint and exit the breakpoint dialog box you are using.
- If you have chosen to update the following views, make sure that you enable the automatic updating of each view:
 - Call Stack view
 - Locals view
 - Watch view.

Breakpoints that have actions assigned to them include the +Act suffix on the breakpoint type shown in the Break/Tracepoints view. Figure 12-4 shows an example of a breakpoint that has an action assigned to it.

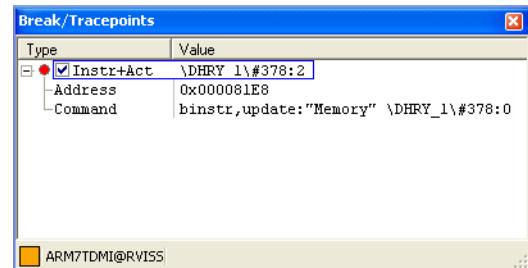


Figure 12-4 Breakpoint with an action assigned

See also

- Setting an unconditional breakpoint with specific attributes* on page 11-16
- Editing a breakpoint* on page 11-25
- Copying a breakpoint* on page 11-28
- Toggling automatic updates of the Locals view* on page 13-18
- Toggling automatic updates of the Call Stack view* on page 13-64
- Toggling automatic updates of the Watch view* on page 13-70
- Displaying information in a user-defined window* on page 13-87.

12.2.2 Updating all windows and views

To update all windows and views when a breakpoint activates:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed, shown in Figure 12-1 on page 12-4.
3. Click **Add** in the Actions group. The Add Action dialog box is displayed, shown in Figure 12-2 on page 12-4.
4. Select the **Update Window** action. This is the default.
5. Select All from the list of windows.
6. Click **OK** to close the Add Action dialog box. The action is added to the breakpoint action list, shown in Figure 12-3 on page 12-5.
7. Click **OK** to set the breakpoint and exit the breakpoint dialog box you are using.

See also

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28.

12.3 Displaying user-defined messages when a breakpoint activates

You can display your own message when a breakpoint activates. The message can be output to:

- the **Cmd** tab of the Output view
- to a user-defined window
- a user-defined file.

Note

You can display only basic messages using this method. To display more complex messages, you can use macros.

To display user-defined messages when a breakpoint activates:

- Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
- Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed, shown in Figure 12-5.



Figure 12-5 Breakpoint behavior controls

- Click **Add** in the Actions group. The Add Action dialog box is displayed.
- Select the **Output Message** action, shown in Figure 12-6.

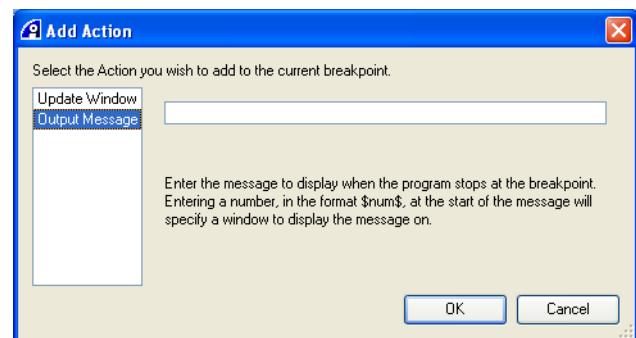


Figure 12-6 Add Action dialog box

- Enter the message that you want displayed when this breakpoint activates. The message can have one of the following formats:
 - message_text* to display the message in the **Cmd** tab of the Output view
 - \$windowid\$message_text* to display the message in a user-defined window
 - \$fileid\$message_text* to write the message to a file.

For example, enter \$100\$this is a message to display the text this is a message in the user-defined window that has an ID of 100.

6. Click **OK** to close the Add Action dialog box. The action is added to the breakpoint action list, shown in Figure 12-7.

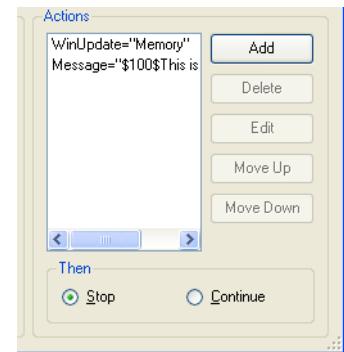


Figure 12-7 The breakpoint action list

7. Click **OK** to set the breakpoint and exit the breakpoint dialog box you are using.

See also:

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- *Toggling automatic updates of the Locals view* on page 13-18
- *Displaying information in a user-defined window* on page 13-87
- *Saving information to a user-defined file* on page 13-90.

12.4 Setting the execution behavior for a breakpoint

By default, a breakpoint stops execution when it is activated. You can change this so that execution automatically continues after the breakpoint activates, and any other actions assigned to the breakpoint are completed.

To set the execution behavior for a breakpoint:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed, shown in Figure 12-8.



Figure 12-8 Breakpoint behavior controls

3. In the Actions, Then group, select either **Stop** (the default) or **Continue** for the execution behavior that is required when the breakpoint activates.
4. Click **OK** to set the breakpoint and exit the breakpoint dialog box you are using.

See also:

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Example of breakpoint behavior* on page 12-27.

12.5 Setting breakpoints that test for hardware input triggers

You can set input trigger tests for hardware breakpoints. The tests match hardware-supported input tests, which depend on the debug target you are using.

See also:

- *Setting a hardware input trigger test*
- *Resetting a hardware input trigger test* on page 12-11
- *Hardware Support settings for RVIS*S on page 12-11
- *Hardware Support settings for an ARM architecture-based target* on page 12-11.

12.5.1 Setting a hardware input trigger test

To set a hardware input trigger test for a breakpoint:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed. Figure 12-9 shows an example:

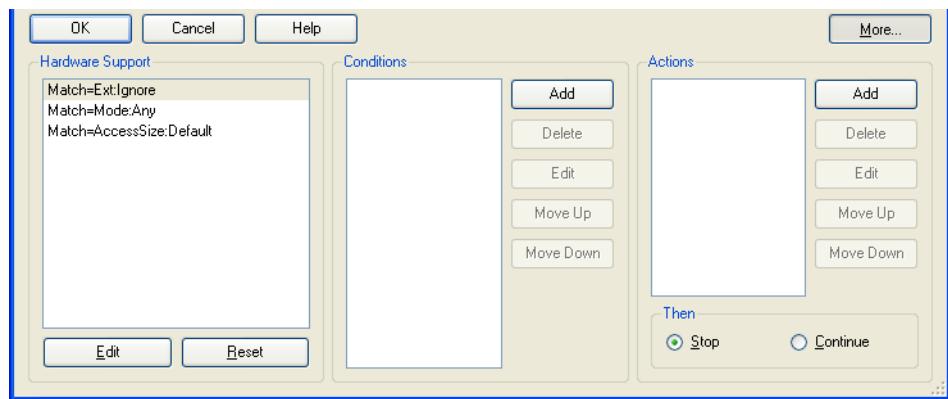


Figure 12-9 Breakpoint behavior controls

3. In the Hardware Support group, select the required input trigger test from the Hardware Support list. The tests available depend on your debug target.
4. Click **Edit** in the Hardware Support group:
 - where a test has a predefined list of options, a Select Hardware Support Value dialog box is displayed, showing the values that you can assign to the selected test
 - where a test requires that you enter a value, a prompt dialog box is displayed, so that you can enter the required value.
5. Specify the value for the test in the dialog box.
6. Click **OK** to close the dialog box. The test in the Hardware Support list is updated with the value you specified.
7. Click **OK** to set the breakpoint and close the breakpoint dialog box.

See also

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Hardware Support settings for RVIS*S on page 12-11

- *Hardware Support settings for an ARM architecture-based target.*

12.5.2 Resetting a hardware input trigger test

To reset a hardware input trigger test:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed. Figure 12-9 on page 12-10 shows an example.
3. Select the test that you want to reset.
4. Click **Reset** in the Hardware Support group to restore value of the test to the default settings.

12.5.3 Hardware Support settings for RVISS

If you are using *RealView ARMulator® ISS* (RVISS), hardware breakpoints have a single Hardware Support group option, **HWPassCount**.

This is initially set to **Off**, but you can set this to an integer value.

See also

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Combining hardware and software pass counts* on page 12-15.

12.5.4 Hardware Support settings for an ARM architecture-based target

Where your debug target supports breakpoint tests in hardware, they can be managed and edited using this group. If enabled, the display lists currently available tests, for example for an ARM® architecture-based target:

AccessSize Supports testing **MAS** signals from the processor. This enables you to test the size of *data bus* activity. The options available depend on the target processor. For example, the options available for an ARM940T™ are:

- Default
- 8-bit
- 16-bit
- 32-bit
- 8/16-bit
- 8/32-bit.

Ext Supports hardware breakpoints that depend on some external condition. This test is not supported on some processors (for example, the ARM1136JF-S™). Where supported, the options available depend on the target processor. For example, the options available for an ARM940T are:

- Ignore
- Low
- High.

Mode Supports testing **nTrans** signals from the processor. This enables you to test the *data not translate* signal to differentiate access between a User mode and a privileged mode. The options available depend on the target processor. For example, the options available for an ARM940T are:

- Any
- Privileged
- User.

12.6 Setting a breakpoint that activates after a number of passes

You can delay the activation of a breakpoint for a specified number of hits.

See also:

- *Procedure when using the generic breakpoint dialog box*
- *Procedure when using the Simple Break if X, N times dialog box* on page 12-14
- *Combining hardware and software pass counts* on page 12-15.

12.6.1 Procedure when using the generic breakpoint dialog box

To delay the activation of a breakpoint for a specified number of hits:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed. Figure 12-10 shows an example:



Figure 12-10 Breakpoint behavior controls

3. Click **Add** in the Conditions group. The Add Condition dialog box is displayed. Figure 12-11 shows an example:

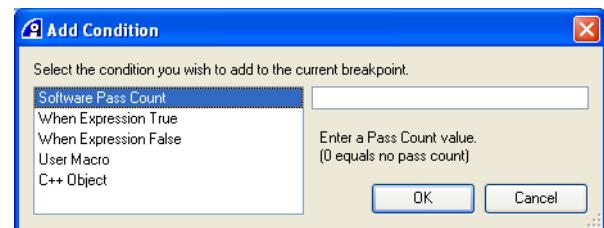


Figure 12-11 Add Condition dialog box

4. Select the **Software Pass Count** condition.
5. Enter the required number of hits before the breakpoint is to activate. For example, if you enter a value of five, then the breakpoint does not activate until hit six.
6. Click **OK** to close the Add Condition dialog box. The condition is added to the breakpoint conditions list.
7. Click **OK** to set the conditional breakpoint, and close the breakpoint dialog box you are using.

The breakpoint is identified by a yellow disc.

Breakpoints that have conditions assigned to them include the +Qual suffix on the breakpoint type shown in the Break/Tracepoints view. Figure 12-12 shows an example of a breakpoint that has a condition assigned to it.

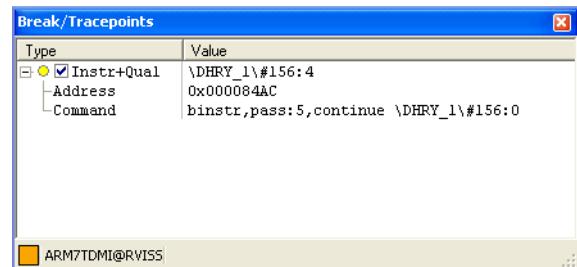


Figure 12-12 Breakpoint with a condition assigned

— Note —

If you are using a debug target that supports it, a pass count can be used with a hardware breakpoint.

See also

- [Unconditional and conditional breakpoints](#) on page 11-6
- [Breakpoint icons and color coding](#) on page 11-9
- [Setting an unconditional breakpoint with specific attributes](#) on page 11-16
- [Editing a breakpoint](#) on page 11-25
- [Copying a breakpoint](#) on page 11-28
- [Resetting breakpoint pass counters](#) on page 12-17
- [Setting breakpoints that test for hardware input triggers](#) on page 12-10.

12.6.2 Procedure when using the Simple Break if X, N times dialog box

To set a breakpoint with the Simple Break if X, N times dialog box:

1. Select **Debug** → **Breakpoints** → **Conditional** → **Break if X, N times...** from the Code window main menu. The Simple Break if X, N times dialog box is displayed. Figure 12-13 shows an example:

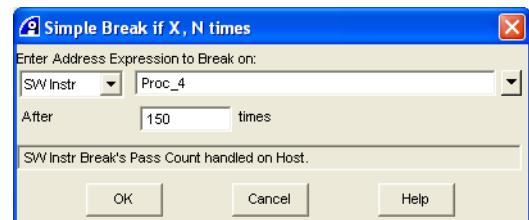


Figure 12-13 Simple Break if X, N times dialog box

This dialog box provides a quick way of creating a breakpoint with the **SW Pass Count** condition qualifier.

2. Choose the type of breakpoint that you want to set. For example, select **HW Instr** to set a hardware execution breakpoint.
3. Specify the location where the breakpoint is to be set. This can be:
 - a specific line number in the source code, with or without a module name prefix
 - a specific address, which can be the address of a variable or function

- a macro that returns an address
- an address range
- a function entry point.

The breakpoint unit activates if the PC equals the corresponding address, or falls within the specified address range. For example, `Proc_4`.

Alternatively, you can click the drop-down arrow to the right of these fields to choose the location from your personal Favorites List, or select from a list of previously-used expressions.

4. In the After field, enter the number of times execution must arrive at the specified address to activate the breakpoint. For example, enter **150** to activate the breakpoint when the hit count reaches 150.
5. Click **OK** to set the breakpoint and close the dialog box.

See also

- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Setting a breakpoint at the entry point to a function* on page 11-40
- *Setting breakpoints from your Favorites List* on page 11-74
- *Resetting breakpoint pass counters* on page 12-17
- *Setting breakpoints that test for hardware input triggers* on page 12-10.

12.6.3 Combining hardware and software pass counts

You can combine hardware and software pass counts to achieve higher count values. If you define both hardware and software pass counts:

1. When the hardware pass count reaches zero, the software pass count is decremented. What happens next depends on your hardware:
 - For RVISS, the hardware pass count remains at zero, so that:

$$\text{total count} = \text{hw_passcount} + \text{passcount}$$
 - Other processors might exhibit the RVISS behavior, or might reset the hardware pass count to the initial value, so that:

$$\text{total count} = (\text{hw_passcount} + 1) * \text{passcount} + \text{hw_passcount}$$
2. When the software pass count reaches zero, the breakpoint activates and the activation count increments. The following example shows the counts for the breakpoint `bexec, hw_pass:3, pass:50 \DHRY_1\#70:0` on an RVISS target:
 - Initial state:


```
> dtbreak
S ID      Type          Address        Count    Miscellaneous
--- ---   Exec      0x00008480        0      Pass=50
```
 - State after activation:


```
> dtbreak
S ID      Type          Address        Count    Miscellaneous
--- ---   Exec      0x00008480        1      Pass=0
```

If the breakpoint is in a loop, then activation occurs on hit 53.

See also

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Resetting breakpoint pass counters* on page 12-17
- *Setting breakpoints that test for hardware input triggers* on page 12-10.

12.7 Resetting breakpoint pass counters

If you have assigned pass counters to a breakpoint, then you can reset the software and hardware pass counters to delay activation of the breakpoint again.

To reset the pass counters for a breakpoint:

1. Select **Break/Tracepoints** from the **View** menu. The Break/Tracepoints view is displayed.
2. For the breakpoint that has the pass counter that you want to reset:
 - a. Right-click on the breakpoint to display the context menu.
 - b. Select **Reset Pass Counters** from the context menu.

The pass counter is reset.

See also:

- *Setting a breakpoint that activates after a number of passes* on page 12-13
- *Setting breakpoints that test for hardware input triggers* on page 12-10.

12.8 Setting a breakpoint that depends on the result of an expression

You can delay the activation of a breakpoint until an expression is either true or false.

See also:

- *Procedure when using the generic breakpoint dialog box*
- *Procedure when using the Simple Break if X, when Y is True dialog box on page 12-19.*

12.8.1 Procedure when using the generic breakpoint dialog box

To delay the activation of a breakpoint until an expression is either true or false:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed. Figure 12-14 shows an example:

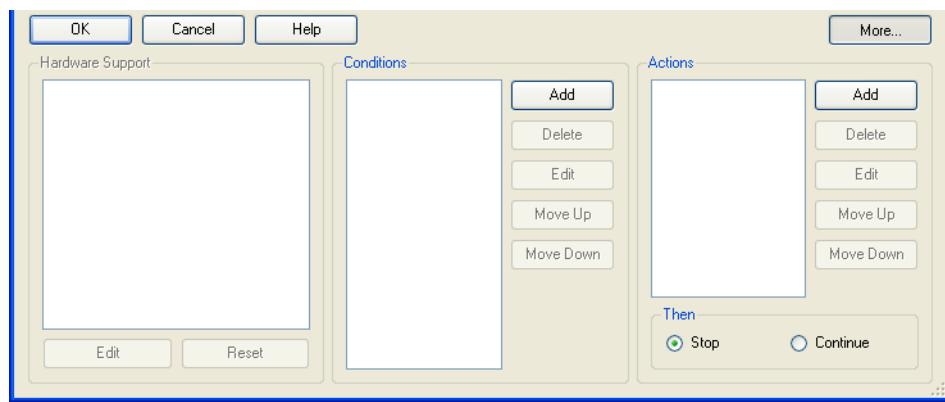


Figure 12-14 Breakpoint behavior controls

3. Click **Add** in the Conditions group. The Add Condition dialog box is displayed. Figure 12-15 shows an example:

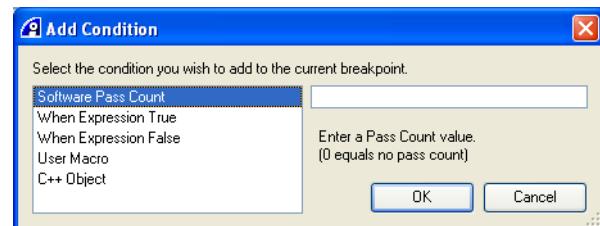


Figure 12-15 Add Condition dialog box

4. Select the required expression condition:
 - **When Expression True**, to delay activation until an expression is true
 - **When Expression False**, to delay activation until an expression is false.
5. Enter the required expression to be tested before the breakpoint is to activate.

———— Note ————

If you specify a variable, then it must be in scope when the breakpoint is hit.

For example, enter `loop_count>=5 & loop_count<10`:

- When used to test for a True condition, the breakpoint activates only when `loop_count` has values from five to nine.

- When used to test for a False condition, the breakpoint activates only when `loop_count` has values from one to four and greater than nine.
6. Click **OK** to close the Add Condition dialog box. The condition is added to the breakpoint conditions list.
 7. Click **OK** to set the conditional breakpoint and close the breakpoint dialog box you are using.
- The breakpoint is identified by a yellow disc.

See also

- *Unconditional and conditional breakpoints* on page 11-6
- *Breakpoint icons and color coding* on page 11-9
- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Constructing expressions* on page 1-14.

12.8.2 Procedure when using the Simple Break if X, when Y is True dialog box

To set a breakpoint with the Simple Break if X, when Y is True dialog box:

1. Select **Debug** → **Breakpoints** → **Conditional** → **Break if X, when Y is True...** from the Code window main menu. The Simple Break if X, N times dialog box is displayed. The Simple Break if X, when Y is True dialog box is displayed. Figure 12-16 shows an example:

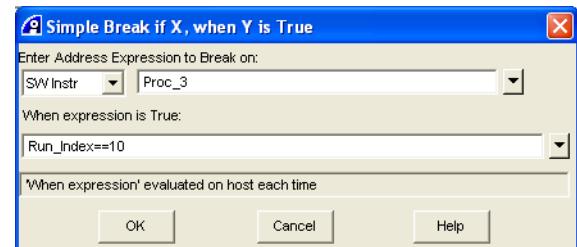


Figure 12-16 Simple Break if X, when Y is True dialog box

This dialog box provides a quick way of creating a breakpoint with the **When Expression True** condition.

2. Choose the type of breakpoint that you want to set. The default is **SW Instr**.
3. Specify the location where the breakpoint is to be set. This can be:
 - a specific line number in the source code, with or without a module name prefix
 - a specific address, which can be the address of a variable or function
 - a macro that returns an address
 - an address range
 - a function entry point.

The breakpoint unit activates if the PC equals the corresponding address, or falls within the specified address range. For example, `Proc_3`.

Alternatively, you can click the drop-down arrow to the right of these fields to choose the location from your personal Favorites List, or select from a list of previously-used expressions.

4. Enter the expression to test (given in C format).

This must give a True or False value as the result. For example, Run_Index==10.

5. Click **OK** to set the breakpoint and close the dialog box.

See also

- *Qualifying breakpoint line number references with module names* on page 11-10
- *Specifying address ranges* on page 11-10
- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Setting a breakpoint at the entry point to a function* on page 11-40
- *Setting breakpoints from your Favorites List* on page 11-74
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Constructing expressions* on page 1-14.

12.9 Setting a breakpoint that depends on the result of a macro

You can create macros that check the values of symbols in your application, or display information about your application during execution. Therefore, by attaching macros to breakpoints you can set up complex conditions or monitor your application at specific points during execution.

The return value from the macro determines whether execution continues or stops when the breakpoint activates.

See also:

- *Attaching a macro as a command qualifier*
- *Attaching a macro as a command argument* on page 12-22
- *Attaching a macro to a global breakpoint* on page 12-22
- *Considerations when using macros with breakpoints* on page 12-22.

12.9.1 Attaching a macro as a command qualifier

To attach a macro to a breakpoint using the `macro` command qualifier:

1. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
2. Click **More...** on the breakpoint dialog box you are using to display the breakpoint behavior controls. Figure 12-14 on page 12-18 shows an example.
3. Click **Add** in the Conditions group. The Add Condition dialog box is displayed, shown in Figure 12-15 on page 12-18.
4. Select the **User Macro** condition.
5. Enter the macro name, including any arguments, that you want to run before the breakpoint is to activate. For example, `test_loop(5,10)`.
6. Click **OK** to close the Add Condition dialog box.
The condition is added to the breakpoint conditions list.
7. Click **OK** to set the conditional breakpoint and close the breakpoint dialog box you are using.
The breakpoint is identified by a yellow disc.

The following example shows a breakpoint command with a macro qualifier:

```
BREAKINSTRUCTION,macro:{test_loop(5,10)} DHRY_1\#149:1
```

Example 12-1 shows the `test_loop` macro. The macro checks when `Run_Index` in `dhrystone` has values between user-specified values. If `Run_Index` has a value within the specified range, then execution stops at the breakpoint.

Example 12-1 Example test_loop macro

```
define /R int test_loop(begin,end)
int begin;
int end;
{
    if ((Run_Index >= begin) & (Run_Index < end))
        return 0; //stop execution
    else
```

```

    return 1; //continue execution
}
.

```

See also

- [Unconditional and conditional breakpoints](#) on page 11-6
- [Breakpoint icons and color coding](#) on page 11-9
- [Setting an unconditional breakpoint with specific attributes](#) on page 11-16
- [Editing a breakpoint](#) on page 11-25
- [Copying a breakpoint](#) on page 11-28
- [Loading user-defined macros](#) on page 16-9
- [Example of breakpoint behavior](#) on page 12-27
- the following in the *RealView Debugger Command Line Reference Guide*:
 - [Macro language](#) on page 1-10 for details of the macro syntax.

12.9.2 Attaching a macro as a command argument

To attach a macro as an argument to a breakpoint, you must use the appropriate CLI command. For example, to attach the macro `test_loop(5,10)` to a software breakpoint at line 149 in `dhrystone_1.c` of the Dhrystone project, enter the command:

```
BREAKINSTRUCTION DHRY_1\#149:1 ; test_loop(5,10)
```

This sets a conditional breakpoint, which is identified by a yellow disc.

See also

- [Unconditional and conditional breakpoints](#) on page 11-6
- [Breakpoint icons and color coding](#) on page 11-9
- [Setting an unconditional breakpoint with specific attributes](#) on page 11-16
- [Editing a breakpoint](#) on page 11-25
- [Copying a breakpoint](#) on page 11-28
- [Loading user-defined macros](#) on page 16-9
- [Example of breakpoint behavior](#) on page 12-27
- the following in the *RealView Debugger Command Line Reference Guide*:
 - [Macro language](#) on page 1-10 for details of the macro syntax.

12.9.3 Attaching a macro to a global breakpoint

A macro can be invoked as an action associated with a global breakpoint, for example:

```
BGLOBAL,enable IRQ ; my_macro()
```

Note

You must use the `BGLOBAL` command if you want to assign a macro to a global breakpoint.

12.9.4 Considerations when using macros with breakpoints

When you set a breakpoint, you can also associate a macro with that breakpoint to set up complex break conditions. When you attach a macro to a breakpoint, it acts as a condition. For example, you can test your program variables and decide how the breakpoint behaves when it activates.

The frequency that a macro runs when the breakpoint activates depends on the order it appears with other conditions.

You can use conditional statements in your macro to change the execution path when the breakpoint is activated depending on variables on the debug target system or on the host. This enables you to control program execution during your debugging session or when there is no user intervention.

You can also use high-level expressions in macros. Combining these conditional statements and expressions enables you to patch your source program.

Breakpoint macros can be used to fill out stubs, such as I/O handling, and also to simulate complex hardware.

— Note —

You cannot use the GO, GOSTEP, STEPINSTR, STEPLINE, STEP0, or STEPOINSTR commands in a macro that is attached to a breakpoint. If any of these commands are present in such a macro, the following messages are displayed:

```
Error: Cannot perform operation - thread is running.  
Error: E0081: Runtime error in macro.
```

RealView Debugger recognizes several predefined macros containing commonly used functions. These macros can also be attached to breakpoints. However, if you are attaching a macro that you create yourself, then you must first load it into RealView Debugger.

Controlling breakpoint behavior with macro return values

The macro return value enables you to control what action RealView Debugger takes when a breakpoint is activated:

- If the macro returns a nonzero value, RealView Debugger continues program execution. Any qualifiers and actions that appear after the macro are not processed, and the breakpoint activation is not recorded.
- If a macro returns a value of zero, RealView Debugger stops program execution. Any actions that are assigned to the macro are performed.

— Note —

This behavior might be different if you assign additional condition qualifiers to the breakpoint.

See also

- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Loading user-defined macros* on page 16-9
- *Example of breakpoint behavior* on page 12-27
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Macro language* on page 1-10 for details of the macro syntax.

12.10 Setting a breakpoint on a specific instance of a C++ class

You can set a breakpoint that activates when a specific instance of C++ class is accessed.

To set a breakpoint that activates for a C++ `this` object:

1. Position the cursor at a point in your source after the object is defined.
2. Right-click on the line of source to display the context menu.
3. Select **Run to Here** from the context menu to run to the current line of source.
4. Open the Create Breakpoint or Edit Breakpoint dialog box as appropriate.
5. Click **More...** on the breakpoint dialog box you are using. The breakpoint behavior controls are displayed. Figure 12-14 on page 12-18 shows an example.
6. Click **Add** in the Conditions group. The Add Condition dialog box is displayed, shown in Figure 12-15 on page 12-18.
7. Select the **C++ Object** condition.
8. Enter the instance of the C++ class to test.
9. Click **OK** to close the Add Condition dialog box. The condition is added to the breakpoint conditions list.
10. Click **OK** to set the conditional breakpoint and close the breakpoint dialog box you are using.

The breakpoint is identified by a yellow disc.

See also:

- *Example of setting a breakpoint on a C++ class* on page 12-25
- *Unconditional and conditional breakpoints* on page 11-6
- *Breakpoint icons and color coding* on page 11-9
- *Setting an unconditional breakpoint with specific attributes* on page 11-16
- *Editing a breakpoint* on page 11-25
- *Copying a breakpoint* on page 11-28
- *Example of setting a breakpoint on a C++ class* on page 12-25.

12.10.1 Example of setting a breakpoint on a C++ class

The source in Example 12-2 is used to show how to set a breakpoint for an instance of a C++ class.

Example 12-2 Example C++ source

```
#include <iostream>
using namespace std;

class Integer {
public:
    Integer(int initVal) : intValue(initVal) {}
    int getValue() { return intValue; }
    void setValue(int newValue) { intValue = newValue; }

private:
    int intValue;
};

int main() {
    Integer valOne(0);
    Integer valTwo(0);
    Integer valThree(0);
    valOne.setValue(1);
    cout << "value = " << valOne.getValue() << "\n";
    valTwo.setValue(2);
    cout << "value = " << valTwo.getValue() << "\n";
    valThree.setValue(3);
    cout << "value = " << valThree.getValue() << "\n";
    return(0);
}
```

Do the following:

1. Copy the source in Example 12-2 to a file called integers.cpp.
2. Build the image, for example:
armcc -g -O3 --no_inline --no_multifile integers.cpp -o integers.axf
3. Connect to an ARM architecture-based target.
4. Load the integers.axf image.
5. Right-click on the line valOne.setValue(1) to display the context menu.
6. Select **Run to Here** from the context menu. The image runs to this line and stops.
7. Enter the following command to set a software instruction breakpoint:
binstr,obj:valTwo \INTEGERS\#8:32
8. Click **Run** on the Debug toolbar. Execution stops for the valTwo object, shown in Figure 12-17 on page 12-26.

The screenshot shows a debugger's code editor window for a file named 'integers.cpp'. The 'Disassembly' tab is selected. The code is as follows:

```

2 using namespace std;
3
4 class Integer {
5 public:
6     Integer(int initVal) : intValue(initVal) {}
7     int getValue() { return intValue; }
8     void setValue(int newValue) { intValue = newValue; } // Breakpoint
9
10 private:
11     int intValue;
12 }
13
14 int main() {
15     Integer valOne(0);
16     Integer valTwo(0);
17     Integer valThree(0);
18
19     valOne.setValue(1);
20     cout << "value = " << valOne.getValue() << "\n";
21 }

```

A red arrow-shaped breakpoint icon is positioned next to the line 8 code 'intValue = newValue;'.

Figure 12-17 Execution stopped on an instance of a C++ object

Also, the following messages are displayed in the **Cmd** tab:

Stopped at 0x0000E1DC due to SW Instruction Breakpoint
 Stopped at 0x0000E1DC: INTEGERS\Integer::setValue Line 8:32

9. Click **Run** again. Execution continues to the end.

12.11 Example of breakpoint behavior

You can control the behavior of a breakpoint by assigning one or more condition qualifiers and actions to that breakpoint. The order that you add the condition qualifiers and actions determines the order they are processed by RealView Debugger. However, actions are performed only when the result of all specified condition qualifiers is True, even if you specify any actions before a condition qualifier.

To demonstrate the breakpoint behavior using the Dhystone image:

1. Connect to your debug target.
2. Load the `dhystone.axf` image.
3. Set a software breakpoint in `dhrystone_1.c`, at column 2 of line 153.

Although you can use the RealView Debugger GUI to set breakpoints, it is easier to understand the interaction of qualifiers and actions using CLI commands.

Enter the following CLI command:

```
breakinstruction,qualifiers \DHRY_1\#153:0
```

For *qualifiers* use the following (see the command qualifiers in Table 12-1 on page 12-28, for each command qualifier variation):

- A macro condition qualifier that specifies a macro to view a symbol, for example:

```
define /R int viewSymbol()
{
    $printsymbol DHRY_1\clock_t$;
    return (0); // 0 - stop execution at the breakpoint
                // >=1 - continue execution
}
```

- A pass count condition qualifier that specifies five passes before the breakpoint is to be activated.
- An action qualifier that prints the message `Actions performed.`

4. Run the image, enter **10** when prompted for the number of runs.
5. If you are using the RealView Debugger GUI, click the **Cmd** tab in the Output view to view the results.
6. Repeat steps 3 to 6 for each command qualifier variation, and change the macro return value as indicated.

See also:

- *Summary of breakpoint behavior* on page 12-28
- *Using a macro as an argument to a break command* on page 12-29
- *Chapter 16 Using Macros for Debugging*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 2 *RealView Debugger Commands*.

12.11.1 Summary of breakpoint behavior

Table 12-1 summarizes the breakpoint behavior for various combinations of the command qualifiers.

Table 12-1 Breakpoint behavior with multiple condition qualifiers and actions

Command qualifiers	Macro return value	Macro executed	Message printed	Breakpoint details recorded
,message:{ "Actions performed"}, passcount:5			After five passes	After five passes
,passcount:5,message:{ "Actions performed"}			After five passes	After five passes
,message:{ "Actions performed"}, ,passcount:5,continue			After five passes	Never
,passcount:5,message:{ "Actions performed"}, ,continue			After five passes	Never
,macro:{viewSymbol()},passcount:5 ,message:{ "Actions performed"}	0 (stop)	After every pass	After five passes	After five passes
,passcount:5,macro:{viewSymbol()} ,message:{ "Actions performed"}	0 (stop)	After five passes	After five passes	After five passes
,macro:{viewSymbol()},passcount:5 ,message:{ "Actions performed"},continue	0 (stop)	After every pass	After five passes	Never
,passcount:5,macro:{viewSymbol()} ,message:{ "Actions performed"},continue	0 (stop)	After five passes	After five passes	Never
,macro:{viewSymbol()},passcount:5 ,message:{ "Actions performed"}	nonzero (continue)	After every pass	Never	Never
,passcount:5,macro:{viewSymbol()} ,message:{ "Actions performed"}	nonzero (continue)	After five passes	Never	Never
,macro:{viewSymbol()},passcount:5 ,message:{ "Actions performed"},continue	nonzero (continue)	After every pass	Never	Never
,passcount:5,macro:{viewSymbol()} ,message:{ "Actions performed"},continue	nonzero (continue)	After five passes	Never	Never

Messages similar to the following are displayed when a breakpoint is recorded:

```
Stopped at 0x00008490 due to SW Instruction Breakpoint
Stopped at 0x00008490: DHRY_1\main Line 153
```

From Table 12-1 you can see that:

- The details of the breakpoint are never recorded when the breakpoint continues. That is when the return value of a macro is nonzero, or if a continue action is assigned.
- When the return value of a macro is nonzero, any actions are never performed, so the Actions performed message is never printed. In addition, any condition qualifiers that are specified after the macro that returns nonzero are never tested. Therefore, when using macros with breakpoints, make sure that you position the macros appropriately, especially if any return a nonzero value.

- The frequency that a macro is executed depends on whether it appears before or after other qualifiers.

Note

The `continue` action qualifier and a macro nonzero value result in different behavior. Although both inhibit the display of the breakpoint details, actions are never performed when a macro returns a nonzero value.

12.11.2 Using a macro as an argument to a break command

You can optionally specify a macro as an argument at the end of a `break` command. Any macro that is specified in this way is treated as being specified last in the command qualifier list. For example:

```
breakinstruction,passcount:5,message:{"Actions performed"} \DHRY_1\#153:0 ;viewSymbol()
```

This command gives the same behavior where the command qualifiers are in the following order (see Table 12-1 on page 12-28):

```
,passcount:5,macro:{viewSymbol()},message:{"Actions performed"}
```

Because a macro argument is treated last in the command qualifier list, if you also specify a macro as a command qualifier, then execution of the macro argument depends on the result returned by the macro qualifier. For example, you might define a second macro, such as:

```
define /R int viewValue()
{
    $printf "Int_1_Loc: %d", Int_1_Loc$;
    return (0); // 0 - stop execution at the breakpoint
                // >=1 - continue execution
}
.
```

Specify this macro as a command argument, and the `viewSymbol()` macro as a command qualifier, for example:

```
breakinstruction,passcount:5,macro:{viewSymbol()},message:{"Actions performed"} (I
A)\DHRY_1\#153:2 ;viewValue()
```

The `viewValue()` macro runs only if `viewSymbol()` returns a value of zero. In addition, if `viewValue()` returns a nonzero value, then the `Actions performed` message is not displayed, and the breakpoint details are not recorded.

See also

- Chapter 16 Using Macros for Debugging*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 2 *RealView Debugger Commands*.

Chapter 13

Examining the Target Execution Environment

This chapter describes how to monitor your program during execution using views in the RealView® Debugger Code window. It contains the following sections:

- *About examining the target execution environment* on page 13-3
- *Finding a function in your code* on page 13-7
- *Displaying function information from the Symbols view* on page 13-10
- *Displaying the list of variables in an image* on page 13-11
- *Viewing variables for the current context* on page 13-14
- *Displaying information for a variable* on page 13-19
- *Viewing C++ classes* on page 13-22
- *Viewing registers* on page 13-25
- *Viewing semihosting controls for DSTREAM or RealView ICE JTAG connections* on page 13-35
- *Viewing semihosting controls for RVISS targets* on page 13-37
- *Viewing memory contents* on page 13-39
- *MMU page tables views* on page 13-46
- *Managing the display of memory in the Memory view* on page 13-50
- *Viewing the Stack* on page 13-58
- *Viewing the Call Stack* on page 13-63
- *Setting watches* on page 13-67
- *Viewing watches* on page 13-70
- *Viewing statistics for RVISS targets* on page 13-74
- *Viewing the RVISS map related statistics in RealView Debugger* on page 13-81
- *Saving memory contents to a file* on page 13-83

- *Comparing target memory with the contents of a file* on page 13-85
- *Displaying information in a user-defined window* on page 13-87
- *Saving information to a user-defined file* on page 13-90
- *Displaying a list of open user-defined windows and files* on page 13-93.

13.1 About examining the target execution environment

RealView Debugger provides various features that enable you to examine parts of your target when it stops at specific points.

You can change the values of variables, registers, memory contents, and watches, to influence how the target subsequently executes.

See also:

- *Functions and variables*
- *C++ classes*
- *Registers* on page 13-4
- *Memory* on page 13-4
- *The Stack* on page 13-4
- *The Call Stack and Locals* on page 13-5
- *Watch* on page 13-5
- *Saving a memory area to a file* on page 13-6
- Chapter 14 *Altering the Target Execution Environment*.

13.1.1 Functions and variables

You can obtain a list of the functions and variables in your image with the Symbols view. The Symbols view also enables you to:

- locate a function
- locate a variable
- perform various operations on a function or variable, such as set a breakpoint.

Variables in scope for the current execution context

You can also view the variables that are in scope at the current execution context. You use the following tabs in the Locals view to do this:

- **Locals** tab, for local variables
- **Statics** tab, for static variables
- **This** tab, for C++ this objects.

See also

- *Finding a function in your code* on page 13-7
- *Displaying the list of variables in an image* on page 13-11
- *Viewing variables for the current context* on page 13-14
- Chapter 10 *Changing the Execution Context*.

13.1.2 C++ classes

If your application is built using C++ source files, then you can view details of the C++ classes with the Classes view.

See also

- *Viewing C++ classes* on page 13-22.

13.1.3 Registers

The Registers view displays the contents of processor registers and enables you to change those contents. Where appropriate, the view shows registers using enumerations to make it easier to read the details, and enables you to enter new values in this format. The Registers view updates the register values to correspond to the current program status each time the target processor stops.

The registers that are visible depend on your debug target, that is your processor and the debug interface. See your processor hardware documentation for details on processor-specific statistics. For information on different debug interfaces, see the appropriate documentation.

Defining new registers

RealView Debugger has built-in awareness of core registers and other standard registers for different processor families. These are displayed in the Registers view. However, you can define new ASIC registers in the target configuration settings. When configured, user-defined registers can be displayed in the Registers view in the same way as standard registers.

See also

- *Viewing registers* on page 13-25
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

13.1.4 Memory

The Memory view enables you to view and change the contents of memory. When you open the Memory view from the **View** menu, the view is empty because no starting address is specified. When you enter a start address, the Memory view shows the current state of the target memory starting at that location. Memory values are updated to correspond to the current program status each time your program stops.

See also

- *Viewing memory contents* on page 13-39.

13.1.5 The Stack

The stack, or run-time stack, is an area of memory used to store function return information and local variables. As each function is called, a record is created on the stack. The record includes traceback details and any local variables. At this point these arguments and local variables become available to RealView Debugger, and you can access them through the Code window.

When the function returns, the area of the stack occupied by that function is made available for the next function call.

In a typical memory-managed ARM processor, the memory map contains the following regions:

- a large area of application memory starting at the lowest address (code and static data)
- an area of memory used to satisfy program requests, the heap, that grows upwards from the top of the application space
- a dynamic area of memory for the stack which grows downwards from the top of memory.

You can view the stack with the Stack view, which enables you to:

- Monitor the contents of the stack as raw memory, and to make changes to those settings. The Stack view shows the contents of the stack at the SP register which is always kept at the top-left of the display area. Use this view to see changes as they happen in the stack.
- Follow the flow of your application through the hierarchical structure by displaying the current state of the stack. This shows you the path that leads from the main entry point to the currently executing function.

The *Stack Pointer* (SP) points to the bottom of the stack.

RealView Debugger can display the calling sequence of any functions that are still in the execution path because their calling addresses are still on the stack. However:

- When a function is off the stack (completes execution), the function information is lost to RealView Debugger.
- If the stack contains a function for which there is no debug information, RealView Debugger might not be able to trace back past it. Make sure all parts of your application are built for debug.

See also

- *Viewing the Stack* on page 13-58
- *ARM® Compiler toolchain Using the Assembler*
- *ARM® Compiler toolchain Using ARM® C and C++ Libraries and Floating-Point Support*
- *ARM® Compiler toolchain Using the Compiler*
- *ARM® Compiler toolchain Compiler Reference*.

13.1.6 The Call Stack and Locals

A call stack is maintained for each processor in your development platform. If you are debugging multithreaded applications, a thread stack is also maintained.

As a program function is called, it is added to the call stack. Similarly, as a function completes execution and returns control normally, it is removed from the call stack. The call stack, therefore, contains details of all functions that have been called but have not yet completed execution.

You can view the call stack with the Call Stack view.

In addition, you can view any local variables, static variables, and C++ `this` objects that are in scope with the Locals view.

See also

- *Viewing variables for the current context* on page 13-14
- *Viewing the Call Stack* on page 13-63.

13.1.7 Watch

Watches enable you to monitor the values of selected variables when execution stops at specific points. You can view watches in the Watch view.

See also

- *Setting watches* on page 13-67.

13.1.8 Saving a memory area to a file

You can save the contents of a memory area into a file, if required. The contents can be saved in a number of formats. You can subsequently load the file into memory, depending on the format.

See also

- *Saving memory contents to a file* on page 13-83.

13.2 Finding a function in your code

You might want to perform various operations on a function in an image, or view information about a function. If so, then you also want to quickly locate the required function. The Symbols view enables you to quickly locate functions in an image.

See also:

- *Finding public and static functions*
- *Finding functions in the libraries used by an image* on page 13-8
- *Viewing the functions of a module* on page 13-9
- *Performing operations on functions from the Symbols view* on page 13-9.

13.2.1 Finding public and static functions

To find a public or static function in your code:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-1 shows an example:

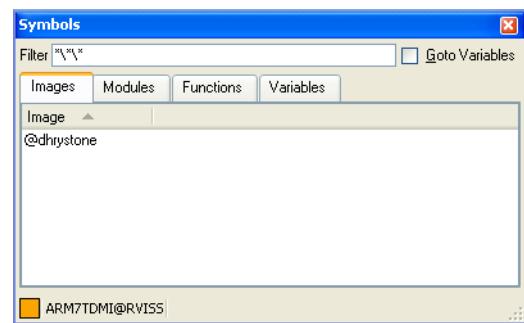


Figure 13-1 Symbols view

2. Click the **Functions** tab. This lists the public and static functions in the image. Figure 13-2 shows an example:

Symbols				
Function Name Address Scope Module Image				
Function Name	Address	Scope	Module	Image
clock	0x000091AC	Public	SYSAPP	@dhystone
exit	0x0000BE8C	Public	STDLIB	@dhystone
fflush	0x0000C020	Public	STDIO	@dhystone
fgetc	0x0000CB90	Public	STDIO	@dhystone
fopen	0x0000BD74	Public	STDIO	@dhystone
fputc	0x0000CBB8	Public	STDIO	@dhystone
free	0x00000000	Public	LCAD1	@dhystone

Figure 13-2 List of functions

3. Right-click on any function entry to display the context menu, and select one or more of the type of functions that you want to display from the context menu:
 - select **Show Publics** to list the public functions (the default)
 - select **Show Statics** to list the static functions (the default).
 If an unwanted function type is already selected, then selecting the corresponding option remove functions of that type from the list.
 Repeat this for each type of function that you want to view or remove from the list.
4. Locate the function that you want to find in your source code.

Note

To reduce the list of functions, specify a filter in the Filter field. For example, to list all functions that begin with the letter P, enter `**\P` or `**\p`.

5. Right-click on the function entry to display the context menu.
6. Select one of the following:
 - **Show in Disassembly**, to display the disassembly view of the function
 - **Show Source**, to view the function in your source code, if available.

The code view changes to show the function.

13.2.2 Finding functions in the libraries used by an image

To locate a function in the libraries used by an image:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-1 on page 13-7 shows an example.
2. Click the **Functions** tab. This lists the functions in the image. Figure 13-2 on page 13-7 shows an example.
3. Right-click on any function entry to display the context menu, and select one or more of the type of functions that you want to display from the context menu:
 - select **Show Publics** to list the public functions
 - select **Show Statics** to list the static functions.

Repeat this for each type of function that you want to view or remove from the list.

4. Right-click on any function entry to display the context menu.
5. Select **Show Library Symbols** from the context menu. The list of functions is expanded to include the library functions.
6. Locate the library function that you want to find.
7. Right-click on the library function entry to display the context menu. You can perform various operations on the functions.

For example, to view the code for a function, select one of the following:

- **Show in Disassembly**, to display the disassembly view of the function
- **Show Source**, to view the function in the source code for your library, if available.

The code view changes to show the function.

Note

To view functions in an ARM library, use the **Show in Disassembly** option.

See also

- *Performing operations on functions from the Symbols view* on page 13-9.

13.2.3 Viewing the functions of a module

You can shorten the list of functions in the **Functions** tab to list only those functions for a specific module in an image. This is especially useful if you have multiple images loaded.

To shorten the list of functions to a specific module in an image:

1. In the **Images** tab of the Symbols view, double-click on the image name. The **Modules** tab is displayed, and lists only the modules for that image, shown in Figure 13-3.

Module Name	Filename	Image
<Global>	<None>	@dhystone
BIGFLT	../../bigflt.c	@dhystone
BTOD_S	../../btod.s	@dhystone
CTYPE	../../ctype.c	@dhystone
D2F_S	../../d2f.s	@dhystone
DCHECK_S	../../dcheck.s	@dhystone
DCHECK1_S	../../dcheck1.s	@dhystone
DHRY_1_S	../../ddhv.s	@dhystone

Figure 13-3 List of modules

Notice that the **Filter** field changes to include the image reference. For example, for the Dhystone image, the filter changes to:

`@dhystone**`

2. Locate the module containing the required function.
3. Double-click on the module name. The **Functions** tab is displayed, and lists only the functions for the chosen module.

Notice that the **Filter** field changes to include the image reference. For example, for the module DHRY_1 in the Dhystone image, the filter changes to:

`@dhystone\DHRY_1*`

If you want to list the functions for multiple modules that have similar names, such as DHRY_1 and DHRY_2, then enter the characters that are common to the required module. For example, enter `dhr*`.

13.2.4 Performing operations on functions from the Symbols view

You can perform various operations on functions using the Symbols view, such as:

- set a breakpoint on the function
- run until the start of the function is reached
- view the source or disassembly for the function
- set the PC to the start of the function
- view type information for the function.

See also

- *Locating a function* on page 5-9
- *Running to the start of a function using the Symbols view* on page 8-9
- *Setting the PC to a function using the Symbols view* on page 10-9
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54
- *Displaying function information from the Symbols view* on page 13-10.

13.3 Displaying function information from the Symbols view

You can view information for a function, which includes the type declaration and where the function is located.

To display information for a function:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-12 on page 13-19 shows an example. If you have loaded multiple images to the target, then all the images are listed.
2. Click the **Functions** tab. This lists the public and static functions in the image. Figure 13-13 on page 13-19 shows an example.
3. Locate the required function.
4. Right-click on the function entry to display the context menu.
5. Select the required option from the context menu:
 - Select **Print Type Information**, to display the type declaration for the function, and where the function is located.
This option issues a PRINTTYPE command.
 - Select **Print Full Information**, to display the address where the function is located, and location in your source code, if any.
This option issues a CEXPRESSION command.

The information is displayed in the **Cmd** tab.

For example, for the function, Func_1 in the Dhystone image, the following information is displayed in the **Cmd** tab:

- for type information:

```
> printtype @dhystone\\DHYR_2\\Func_1
extern Enumeration Func_1(int, int);
-- In module DHYR_2, filename = 'dhry_2.c', starting at line 118
```
- for full information:

```
> cexpr @dhystone\\DHYR_2\\Func_1
Result is: code address 0x00009048 @dhystone\\DHYR_2\\Func_1
Line 118..118 at 0x00009048..0x0000904F {Func_1}
```

See also:

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the CEXPRESSION command.

13.4 Displaying the list of variables in an image

You might want to perform various operations on a variable in an image, or view information about a variable. If so, then you also want to quickly locate the required variable. The Symbols view enables you to quickly locate variables in an image.

See also:

- *Displaying variables*
- *Displaying variables in the libraries used by an image* on page 13-12
- *Viewing the variables for a module* on page 13-13
- *Performing operations on variables from the Symbols view* on page 13-13.

13.4.1 Displaying variables

To display variables in an image:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-4 shows an example. If you have loaded multiple images to the target, then all the images are listed.

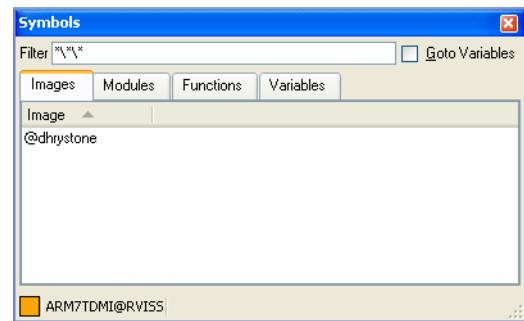


Figure 13-4 Symbols view

2. Click the **Variables** tab. By default, this lists the public and static variables in the image. Figure 13-5 shows an example:

Symbols				
Filter: *%* Goto Variables				
Variable Name	Address	Scope	Module	Image
Arr_1_Glob	0x0000DC78	Public	<Global>	@dhystone
Arr_2_Glob	0x0000DD40	Public	<Global>	@dhystone
Begin_Time	0x0000DC50	Public	<Global>	@dhystone
Bool_Glob	0x0000DC48	Public	<Global>	@dhystone
Ch_1_Glob	0x0000DC4C	Public	<Global>	@dhystone
Ch_2_Glob	0x0000DC4D	Public	<Global>	@dhystone
Dhystones_Per_Second	0x0000DC60	Public	<Global>	@dhystone
End_Time	0x0000DC54	Public	<Global>	@dhystone
Int_Glob	0x0000DC44	Public	<Global>	@dhystone
Microseconds	0x0000DC5C	Public	<Global>	@dhystone

Figure 13-5 List of variables

Note

To reduce the list of variables, specify a filter in the Filter field. For example, to list all variables that begin with the letter C, enter **\C or **\c.

3. Right-click on any variable entry to display the context menu, and select one or more of the type of variables that you want to display from the context menu:
 - select **Show Publics** to list the public variables (the default)

- select **Show Statics** to list the static variables (the default)
 - select **Show Locals** to list the local variables.
- Repeat this for each type of variable that you want to view or remove from the list.
4. Right-click on the variable entry to display the context menu. You can perform various operations on the variables.

See also

- *Performing operations on variables from the Symbols view* on page 13-13.

13.4.2 Displaying variables in the libraries used by an image

To display a list of variables in the libraries used by an image:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-4 on page 13-11 shows an example.
2. Click the **Variables** tab. This lists the variables in the image. Figure 13-5 on page 13-11 shows an example.
3. Right-click on any variable entry to display the context menu, and select one or more of the type of variables that you want to display from the context menu:
 - select **Show Publics** to list the public variables
 - select **Show Statics** to list the static variables
 - select **Show Locals** to list the local variables.
 Repeat this for each type of variable that you want to view or remove from the list.
4. Select **Show Library Symbols** from the context menu. The list of variables is expanded to include the library variables.
5. Locate the library variable that you want to find.
6. Right-click on the variable entry to display the context menu. You can perform various operations on the variables.

See also

- *Performing operations on variables from the Symbols view* on page 13-13.

13.4.3 Viewing the variables for a module

You can shorten the list of variables in the **Variables** tab to list only those variables for a specific module in an image. This is especially useful if you have multiple images loaded.

To shorten the list of variables to a specific module in an image:

- In the **Images** tab of the Symbols view, double-click on the image name. The **Modules** tab is displayed, and lists only the modules for that image, shown in Figure 13-6.

Module Name	Filename	Image
<Global>	<None>	@dhystone
BIGFLT	../../bigflt.c	@dhystone
BTOD_S	../../btod.s	@dhystone
CTYPE	../../ctype.c	@dhystone
D2F_S	../../d2f.s	@dhystone
DCHECK_S	../../dcheck.s	@dhystone
DCHECK1_S	../../dcheck1.s	@dhystone
DHRY_1_S	../../dhry1.s	@dhystone

Figure 13-6 List of modules

Notice that the **Filter** field changes to include the image reference. For example, for the Dhystone image, the filter changes to:

`@dhystone*/*`

- Locate the module containing the required function.
- Select the **Goto Variables** check box.
- Double-click on the module name. The **Variables** tab is displayed, and lists only the variables for the chosen module.

Notice that the **Filter** field changes to include the image reference. For example, for the module DHRY_1 in the Dhystone image, the filter changes to:

`@dhystone\DHRY_1*`

If you want to list the variables for multiple modules that have similar names, such as DHRY_1 and DHRY_2, then change the filter to include wildcards. For example:

`@dhystone\DHRY**`

If you want to apply a filter on the complete list of variables, then enter the characters that are common to the required variables. For example, to list all variables that begin with the letter C, then enter C or c (filtering is not case sensitive).

13.4.4 Performing operations on variables from the Symbols view

You can perform various operations on variables using the Symbols view, such as:

- set a data access breakpoint on a variable
- set a watch point on a variable
- print the value of a variable in hexadecimal or decimal
- view information about a variable.

See also

- Displaying information for a variable* on page 13-19
- Setting breakpoints for memory accesses* on page 11-46
- Setting breakpoints for location-specific data values* on page 11-54
- Setting watches* on page 13-67.

13.5 Viewing variables for the current context

The following sections describe how to view the variables that are in scope for the current context:

- *Displaying local variables for the current context*
- *Displaying static variables for the current context* on page 13-15
- *Displaying C++ this pointers for the current context* on page 13-16
- *Performing timed updates for RealMonitor and OS-aware targets* on page 13-16
- *Formatting the display of values for individual variables* on page 13-17
- *Formatting the display of values for all variables* on page 13-17
- *Toggling automatic updates of the Locals view* on page 13-18
- *Manually updating values in the Locals view* on page 13-18
- *Copying and pasting variables from the Locals view to the Watch view* on page 13-18.

13.5.1 Displaying local variables for the current context

To display the local variables that are in scope for the current context:

1. Display the **Locals** tab in the Locals view:
 - If the view is not already visible, select **Locals** from the **View** menu.
 - If the view is already visible, click the **Locals** tab in the view.

The **Locals** tab is displayed, as shown in Figure 13-7, and lists the local variables that are currently in scope.

Name	Value
*Enumeration Func_1(Capital_Letter, Capital_Letter)	
Ch_1_Par_Val	0x52 'R' <@r3>
Ch_2_Par_Val	0x59 'Y' <@r4>
Ch_1_Loc	0x52 'R' <@r12>
Ch_2_Loc	0x52 'R' <@r14>

Locals Statics This

ARM7TDMI@RVISS

Figure 13-7 Locals tab in the Locals view

2. You can perform various operations on the local variables shown in the tab.

See also

- *Performing timed updates for RealMonitor and OS-aware targets* on page 13-16
- *Formatting the display of values for individual variables* on page 13-17
- *Copying and pasting variables from the Locals view to the Watch view* on page 13-18
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54.

13.5.2 Displaying static variables for the current context

To display the static variables that are in scope for the current context:

1. Select **Locals** from the **View** menu to display the Locals view. Figure 13-8 shows an example:

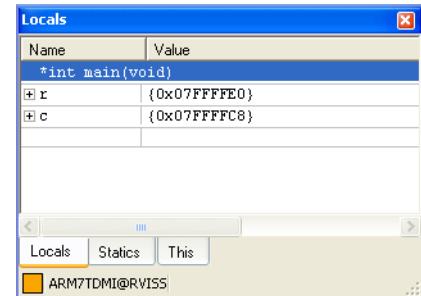


Figure 13-8 Locals tab in the Locals view

2. Click the **Statics** tab in the Locals view. Figure 13-9 shows an example:

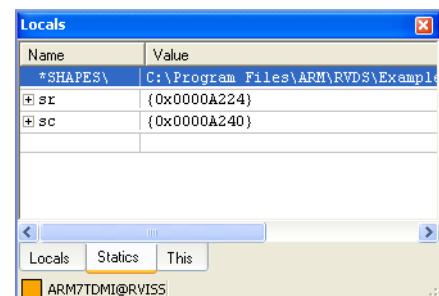


Figure 13-9 Statics tab in the Locals view

3. You can perform various operations on the static variables shown in the tab.

See also

- *Performing timed updates for RealMonitor and OS-aware targets* on page 13-16
- *Formatting the display of values for individual variables* on page 13-17
- *Copying and pasting variables from the Locals view to the Watch view* on page 13-18
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54.

13.5.3 Displaying C++ this pointers for the current context

To display the C++ this pointers that are in scope for the current context:

1. Select **Locals** from the **View** menu to display the Locals view. Figure 13-8 on page 13-15 shows an example.
2. Click the **This** tab in the Locals view. Figure 13-10 shows an example:

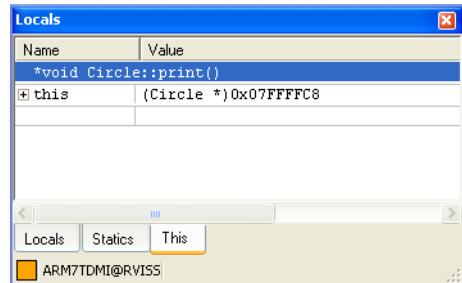


Figure 13-10 This tab in the Locals view

3. You can perform various operations on the C++ this pointers.

See also

- *Performing timed updates for RealMonitor and OS-aware targets*
- *Formatting the display of values for individual variables* on page 13-17
- *Copying and pasting variables from the Locals view to the Watch view* on page 13-18
- *Setting breakpoints for memory accesses* on page 11-46
- *Setting breakpoints for location-specific data values* on page 11-54.

13.5.4 Performing timed updates for RealMonitor and OS-aware targets

If you are using RealMonitor or an OS extension, the Locals view can be updated at a specified time interval during program execution.

To perform a timed update of the Locals view:

1. Connect to a target that is configured to run either:
 - RealMonitor
 - an OS-aware application.
2. Right-click in the Locals view to display the context menu.
3. Select **Timed Update Period...** from the context menu to display the Timed Update Period dialog box. Figure 13-11 shows an example:

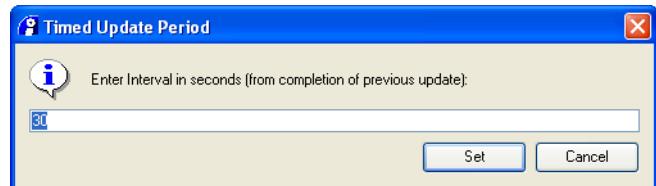


Figure 13-11 Timed Update Period dialog box

4. Enter the interval, in seconds, between window updates. The default is 30 seconds. This value is used only when **Timed Update** is enabled.
5. Right-click in the Locals view to display the context menu.

6. Select **Timed Update** from the context menu.

This option is enabled when:

- RealMonitor is activated
- the target is in RSD mode and where supported by the underlying debug target.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43
- *RealView Debugger RTOS Guide* for details on debugging OS-aware applications.

13.5.5 Formatting the display of values for individual variables

To format the display of values in the **Locals**, **Statics** or **This** tab:

1. Right-click on the name or value of the variable to display the context menu.
For arrays, structures, or C++ objects:
 - if you right-click on the name, then all values associated with that variable are formatted.
 - if you right-click on the value, then only that value is formatted.
2. Select **Format...** from the context menu to display the selection dialog box. Highlight the required format for the expression from the list of available formats.

————— **Note** —————

If a variable contains multibyte characters, you can choose to view it using ASCII, UTF-8, or Locale encoding.

3. Select the required format.
4. Click **OK** to apply the chosen format, and close the dialog box.

13.5.6 Formatting the display of values for all variables

To format the display of all variables:

1. Right-click in the Locals view to display the context menu.
2. Select the formatting option you require from the context menu:

Show char* and char[] as strings

Displays local variables of type `char*` and `char[]` as strings.
This is enabled by default.

Show integers in hex

Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal.
This is enabled by default.

————— **Note** —————

These options do not affect the display of values for any variables that you have formatted individually.

13.5.7 Toggling automatic updates of the Locals view

By default, the Locals view updates automatically. To toggle the automatic update of the Locals view:

1. Right-click in the Locals view to display the context menu.
2. Select **Automatic Update** from the context menu.

Automatic updating refreshes the Locals view as soon as execution stops at a point in your image, such as at a breakpoint.

Freezing the contents of the Locals view

To freeze the contents of the Locals view, disable automatic updating. You can manually update the view contents if required.

Freezing the contents of the Locals view enables you to compare the contents with those in a second Locals view.

See also

- *Updating windows and views when a breakpoint activates* on page 12-4
- *Manually updating values in the Locals view*.

13.5.8 Manually updating values in the Locals view

To manually update the values in the Locals view:

1. Disable automatic updates.
 2. Right-click in the Locals view to display the context menu.
 3. Select **Update Window** from the context menu.
- The values are updated.

See also

- *Toggling automatic updates of the Locals view*.

13.5.9 Copying and pasting variables from the Locals view to the Watch view

To copy a variable from the Locals view and paste it into the Watch view:

1. Select the variable to be copied.
2. Press Ctrl+C to copy the selected variable.
3. Select **Watch** from the **View** menu to display the Watch view.
4. Click on the background in the Watch view.
5. Press Ctrl+V to paste the variable into the Watch view. Be aware that both the variable name and the value are copied.

See also

- *Viewing memory at a variable value by copying and pasting* on page 13-42
- *Setting a watch by copying and pasting* on page 13-68.

13.6 Displaying information for a variable

The following sections describe how to view type information for a variable, or display the contents of a variable:

- *Printing the value of a variable*
- *Displaying type information for a variable* on page 13-20.

13.6.1 Printing the value of a variable

To print the value of a variable:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-12 shows an example. If you have loaded multiple images to the target, then all the images are listed.

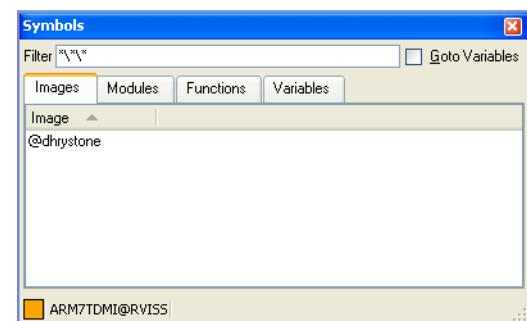


Figure 13-12 Symbols view

2. Click the **Variables** tab. This lists the public and static variables in the image. Figure 13-13 shows an example:

Symbols				
Filter: @dhystone				
Variable Name	Address	Scope	Module	Image
Arr_1_Glob	0x0000DC78	Public	<Global>	@dhystone
Arr_2_Glob	0x0000DD40	Public	<Global>	@dhystone
Begin_Time	0x0000DC50	Public	<Global>	@dhystone
Bool_Glob	0x0000DC48	Public	<Global>	@dhystone
Ch_1_Glob	0x0000DC4C	Public	<Global>	@dhystone
Ch_2_Glob	0x0000DC4D	Public	<Global>	@dhystone
Dhystone_Per_Second	0x0000DC60	Public	<Global>	@dhystone
End_Time	0x0000DC54	Public	<Global>	@dhystone
Int_Glob	0x0000DC44	Public	<Global>	@dhystone
Microseconds	0x0000DC5C	Public	<Global>	@dhystone

Figure 13-13 List of variables

3. Locate the required variable.
4. Right-click on the variable entry to display the context menu.
5. Select the required option from the context menu:
 - select **Print Hex**, to display the value of the variable in hexadecimal
 - select **Print Decimal**, to display the value of the variable in decimal.

The information is displayed in the **Cmd** tab.

These options issue a PRINT command, which is the short form of PRINTVALUE.

For example, for the array Arr_2_Glob in the Dhystone image, the following information is displayed in the **Cmd** tab:

- for values in hexadecimal:


```
> print /h @dhystone\\Arr_2_Glob
0x0000DD40 = {{0x0 <repeats 50 times>} <repeats 8 times>,{0x0,0x0,0x0,
0x0,0x0,0x0,0xA,0x0 <repeats 42 times>},
{0x0 <repeats 50 times>} <repeats 41 times>}
```
- for values in decimal:


```
> print @dhystone\\Arr_2_Glob
0x0000DD40 = {{0 <repeats 50 times>} <repeats 8 times>,{0,0,0,0,0,0,0,
10,0 <repeats 42 times>},{0 <repeats 50 times>} <repeats 41 times>}
```

See also

- *Viewing memory contents* on page 13-39
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the PRINTVALUE command.

13.6.2 Displaying type information for a variable

To display the type information for a variable:

1. Select **Symbols** from the **View** menu to display the Symbols view. Figure 13-12 on page 13-19 shows an example. If you have loaded multiple images to the target, then all the images are listed.
2. Click the **Variables** tab. This lists the public and static variables in the image. Figure 13-13 on page 13-19 shows an example.
3. Locate the required variable.
4. Right-click on the variable entry to display the context menu.
5. Select the required option from the context menu:
 - Select **Print Type Information** to display the type information for the variable. This option issues a PRINTTYPE command.
 - Select **Print Full Information** to display the scope information for the variable and the address where the variable is located. This option issues a PRINTSYMBOL command.

The information is displayed in the **Cmd** tab.

For example, for the array Arr_2_Glob in the Dhystone image, the following information is displayed in the **Cmd** tab:

- for type information:


```
> printtype @dhystone\\Arr_2_Glob
extern int Arr_2_Glob[50][50];
```
- for full information:


```
> printsym @dhystone\\Arr_2_Glob
@dhystone\\Arr_2_Glob: Global int[50][50].
Address = 0x0000DD40 to 0x0001044F
```

See also

- *Viewing memory contents* on page 13-39
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the PRINTSYMBOL and PRINTTYPE commands.

13.7 Viewing C++ classes

If your application includes C++ sources, then you can view the C++ classes with the Classes view.

See also:

- *Displaying the properties of a class or a component of a class*
- *Identifying the components of a class in the Classes view* on page 13-23
- *Finding a class definition in your source code* on page 13-23
- *Operations you can perform on member functions* on page 13-24.

13.7.1 Displaying the properties of a class or a component of a class

To view the C++ classes in an image:

1. Select **Classes** from the **View** menu to display the Classes view. Figure 13-14 shows an example:

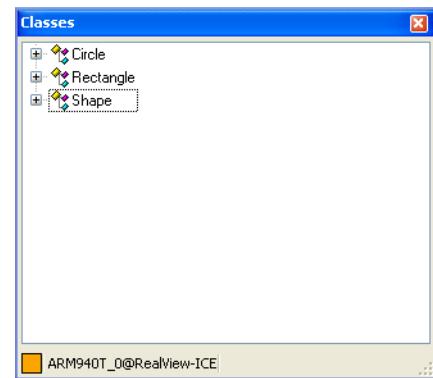


Figure 13-14 Classes view

2. Locate the required class.
3. Expand the class entry to see the components. Figure 13-15 shows an example:

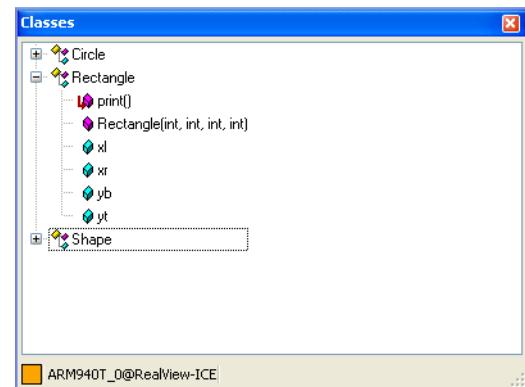


Figure 13-15 Components of class entries in the Classes view

4. Right-click on the class or a component of the class to display the context menu.
5. Select **Properties...** from the context menu to display the Properties dialog box showing the properties of the selected class or component.

See also

- *Identifying the components of a class in the Classes view.*

13.7.2 Identifying the components of a class in the Classes view

Colored icons are used to identify different components within a class (see Figure 13-15 on page 13-22):

**Filled stack + arrow**

The magenta filled stack with an arrow indicates a function that is a declared member of the parent class.



Filled stack The magenta filled stack indicates a member function of the parent class that is both declared and defined. These members are real in that they are called during execution.

You can perform various operations on these functions.

**Hollow stack**

A hollow magenta stack indicates a member function of the parent class that is declared but not defined. These members are virtual in that they are not called during execution.



Filled block The blue filled block indicates a data object that is manipulated by a class function using operators, or methods, defined in the class.

See also

- *Operations you can perform on member functions* on page 13-24.

13.7.3 Finding a class definition in your source code

To find the definition of a class in your source code:

1. Right-click on a chosen class to display the context menu.
2. Select **Find Class Definition...** from the context menu to display the Find in Files dialog box. Figure 13-16 shows an example:

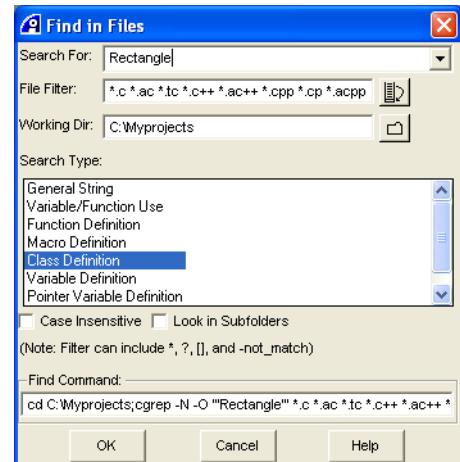


Figure 13-16 Find In Files dialog box

Modify the class details to be searched, if required.

3. Click **OK** to begin the search.

The search results are displayed in the **FileFind** tab of the Output view.

———— **Note** ————

Use this dialog box to locate class definitions when these are not provided by the compiler.

13.7.4 Operations you can perform on member functions

You can perform the following operations on declared and defined member functions:

- locate the source for a member function
- set a breakpoint at the start of a member function.

See also

- *Locating a member function in a C++ class* on page 5-12
- *Setting a breakpoint at the start of a member function in a C++ class* on page 11-43
- *Identifying the components of a class in the Classes view* on page 13-23 for details of the filled stack.

13.8 Viewing registers

The following sections describe the options available when working with registers:

- *Examining registers in the Registers view*
- *Displaying the properties of a register* on page 13-27
- *Creating a customized register view for a connection* on page 13-28
- *Formatting selected registers in the Registers view* on page 13-29
- *Formatting all registers in the Registers view* on page 13-29
- *Copying all registers in the current register view* on page 13-29
- *Copying selected registers in the current register view* on page 13-30
- *Updating the register display* on page 13-30
- *Viewing different registers* on page 13-30
- *Viewing registers for multiple targets* on page 13-31
- *Understanding the register view* on page 13-31
- *Viewing debugger internals* on page 13-33.

Note

If you are using *RealView ARMulator® ISS* (RVISS), then the registers in some ARM processors are incompletely modeled, that is:

- ARM925T™, wait for interrupt
- ARM966E-S™-REV2, TCM register size missing
- ARM946E-S™-REV1, r13 trace PID
- ARM926EJ-S™, r13 context ID writing to the wrong register
- ARM720T™-REV4, does not distinguish trace PID and FCSE r13.

13.8.1 Examining registers in the Registers view

To examine the contents of registers:

1. Connect to your target.

2. Load an image.

For example, load the Dhrystone example image `dhrystone.axf`.

3. Select **Registers** from the **View** menu to display the Registers view as a floating view. The **Core** tab is displayed by default. Figure 13-17 shows an example:

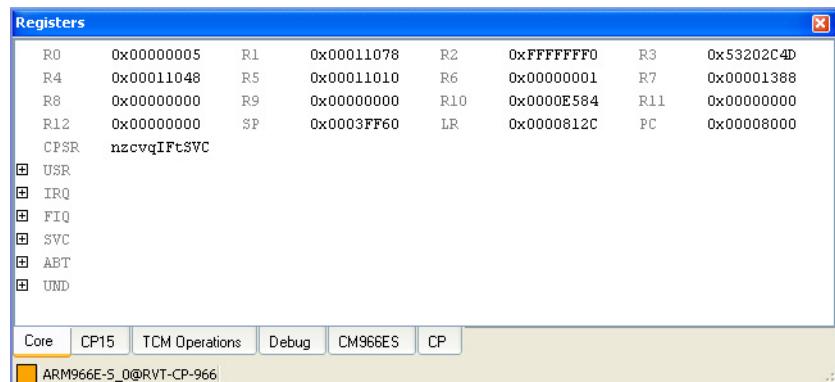


Figure 13-17 Registers view

The Registers view displays tabs appropriate to the:

- Target processor running your image.
Different target processors contain different registers and so the contents of this view change depending on the target you are debugging.
- Debug Interface used to interface to the development platform.
- BCD files assigned to the Debug Configuration.

Demonstrating how the register view changes during execution

The following procedure demonstrates how the register view changes during execution:

1. Follow the procedure described in *Examining registers in the Registers view* on page 13-25.
2. Click the **Locate PC** button on the Debug toolbar to display the source file containing the current scope.
3. Click the **dhry_1.c** tab to view the source file dhry_1.c.
4. Set a simple breakpoint by double-clicking in the gray margin on line 301.
5. Click **Run** to start execution.
6. Enter 5000 when asked for the number of runs. The program starts and then stops when execution reaches the breakpoint at line 301. The yellow arrow and red box marks the location of the PC when execution stops.
7. The contents of the Registers view are updated to show the program status as the target stops, shown in Figure 13-18.

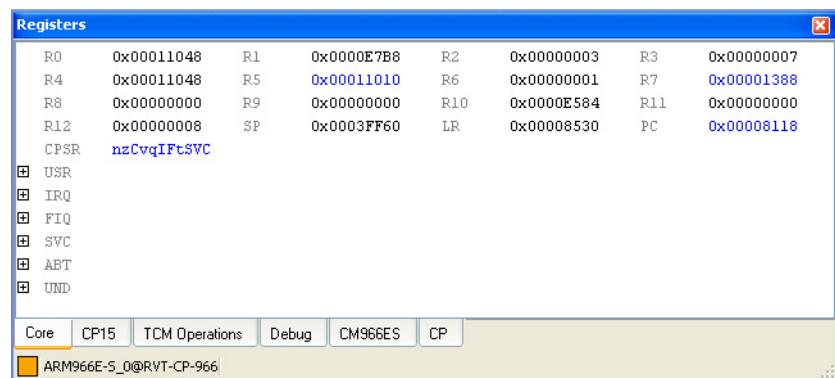


Figure 13-18 Registers view

8. Click the **Step** button to step one line of source. Register values that have changed since the last update are displayed in dark blue, shown in Figure 13-19 on page 13-27.



Figure 13-19 Registers changed since the last update

- Click **Step** again and examine the register values as they change. Register values that changed at the previous update are displayed in pale blue, shown in Figure 13-20.

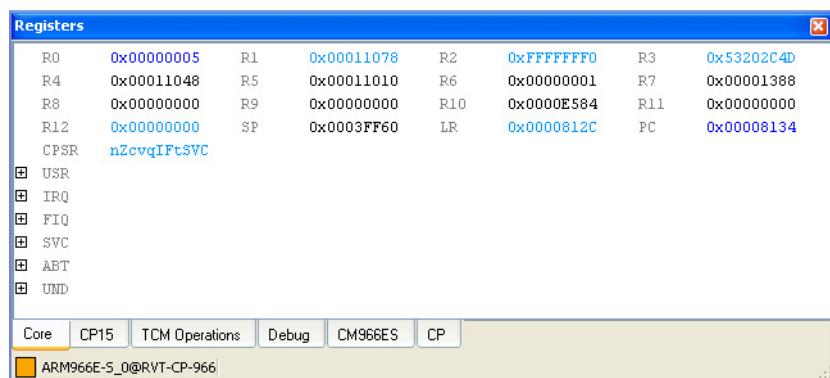


Figure 13-20 Registers changed at the previous update

- Repeat the last step as required.
- Double-click on the red marker disc to clear the breakpoint at line 301.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- *Viewing different registers* on page 13-30
- *Understanding the register view* on page 13-31
- *Viewing debugger internals* on page 13-33.

13.8.2 Displaying the properties of a register

To display the properties of a register:

- Connect to your target.
- Load an image.
For example, load the Dhrystone example image `dhrystone.axf`.
- Select **Registers** from the **View** menu to display the Registers view as a floating view. The **Core** tab is displayed by default. Figure 13-17 on page 13-25 shows an example.
- Click the tab in the Registers view that contains the register of interest.

5. Right-click on the required register to display the context menu.
6. Select **Properties...** from the context menu. The properties dialog box is displayed. Figure 13-21 shows an example:



Figure 13-21 Register properties dialog box

The dialog box shows the RealView Debugger symbol name and the data type for the register. You can use the symbol name in command scripts.

See also

- Chapter 15 *Debugging with Command Scripts*.

13.8.3 Creating a customized register view for a connection

Each tab in the Registers view displays the complete set of registers identified by the name of the tab. For example, the **Core** tab displays the complete set of core registers for a processor. However, you might only be interested in viewing some registers from one or more tabs. RealView Debugger enables you to create your own customized register view, that contains only those registers of interest.

Note

The customized register view is connection-specific. Therefore, if you have multiple connections, you can create a different customized register view for each connection. Only the **User** tab for the connection shown in the Code window is available in the Registers view.

To create your own customized register view:

1. Connect to your target.
2. Load an image.
For example, load the Dhystone example image `dhystone.axf`.
3. Select **Registers** from the **View** menu to display the Registers view as a floating view. The **Core** tab is displayed by default. Figure 13-17 on page 13-25 shows an example.
4. Click the tab in the Registers view that contains the registers of interest.
5. Select the register that you want to copy to your own register view. You can select multiple registers to copy using the standard key operations, if required.
For example, in the **Core** tab select the PC, SP, LR, R0, and R1 registers.
6. Right-click on a selected register, to display the context menu.
7. Select **Copy to User View** from the context menu. The selected registers are copied to your user view.
8. If the **User** tab is not visible, right-click on the white background in the view to display the context menu.
9. Select **Show User View** from the context menu to display the **User** tab containing the registers you have added to the view. Figure 13-22 on page 13-29 shows an example:

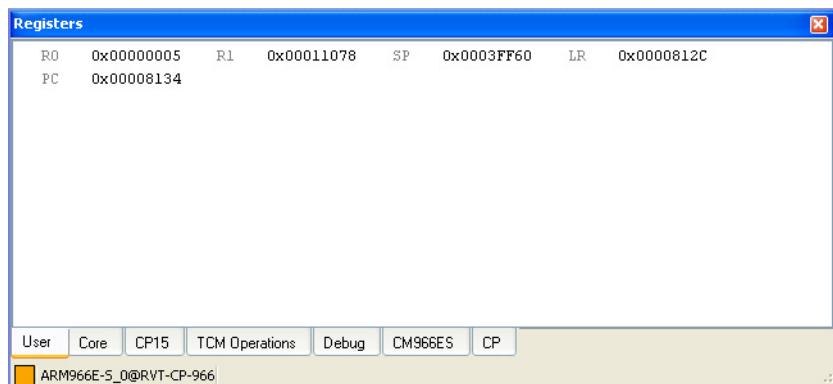


Figure 13-22 Customized register view

10. Repeat this procedure to add more registers from other tabs as required.

Note

Any registers on the **User** tab persist between debugger sessions.

13.8.4 Formatting selected registers in the Registers view

You can change the display format for selected registers without affecting the display format of other registers. To do this:

1. Right-click on a chosen register, for example R3 in the **Core** tab, to display the context menu. This menu also enables you to view the memory at the register contents, and to copy and paste values.
2. Select **Format** to display the format options menu.
3. Select the required format from the format options menu. The Register view refreshes to make sure the value of the register is completely displayed in the new format.

13.8.5 Formatting all registers in the Registers view

To format all registers that are currently displayed:

1. Select all the registers in the current tab. Registers in an unopened register bank are not selected. For example, if only the **USR** register bank on the **Core** tab is open, the registers in the **USR** bank are selected, but the registers in the **IRQ**, **FIQ**, **SVC**, **ABT**, and **UND** banks are not selected.
2. Right-click on any register, to display the context menu.
3. Select **Format** to display the format options menu.
4. Select the required format from the format options menu.

13.8.6 Copying all registers in the current register view

You might want to copy the current register view to take a snapshot of the current values. You can then copy this view, for example into a text editor, so that you can compare the register values.

To copy all registers in the current register view:

1. Right-click on the background in the Registers view to display the context menu.

2. Select **Copy View** from the context menu.

Note

For the CPSR and SPSR registers, both the register name and the setting string are copied.

The registers in a banked register are copied only if that register bank is expanded. Otherwise, only the name of the banked register is copied.

13.8.7 Copying selected registers in the current register view

You might want to copy selected registers in the current register view to take a snapshot of the current values. You can then copy this view, for example into a text editor, so that you can compare the register values.

To copy selected registers in the current register view:

1. Select the registers containing the values that you want to copy.
2. Right-click on a selected register to display the context menu.
3. Select **Copy Value** from the context menu.

Note

Register names are not copied. For the CPSR and SPSR registers, only the setting string is copied.

13.8.8 Updating the register display

To update the register display:

1. Right-click on the background in the Registers view to display the context menu.
2. Select **Update Window** from the context menu.

Note

You can also update the Registers view when a breakpoint activates.

See also

- *Updating windows and views when a breakpoint activates* on page 12-4.

13.8.9 Viewing different registers

The Registers view displays tabs and registers appropriate to your:

- Target processor, that is different target processors contain different registers and so the contents of this view change depending on the target you are debugging.
- Debug Interface, for example DSTREAM or RealView ICE.
- Debug Configuration, if you are using a supplied BCD file or you have defined custom registers in your own BCD file.

In Figure 13-23 on page 13-31, the Registers view shows registers for an ARM940T™ processor and the ARM Integrator™/CP board using RealView ICE.

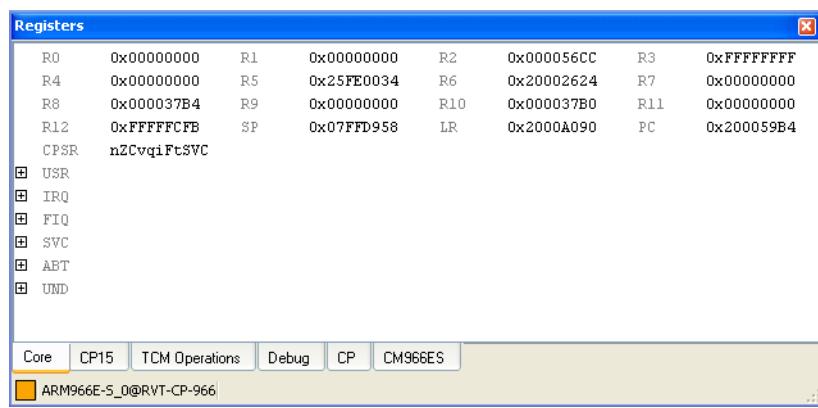


Figure 13-23 Registers for an Integrator/CP and ARM966E-S debug target

In this example, you can see the **CP15** tab and other tabs relating to processor-specific operations, for example **TCM Operations**. These are special registers and are described in full in the processor hardware documentation.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

13.8.10 Viewing registers for multiple targets

If you are licensed to use multiprocessor debugging mode you can access different registers on multiple debug targets at the same time. To do this:

1. Connect to each target of your multiprocessor system.
 2. Display multiple Code windows.
 3. Attach each Code window to a different debug target.
 4. In each Code window, display the registers for the associated target.
- RealView Debugger enables you to set up multiple Registers views with different formatting options for each.

See also

- *Connecting to multiple targets* on page 3-46
- *Displaying multiple Code windows* on page 7-8
- *Attaching a Code window to a connection* on page 7-10
- *Examining registers in the Registers view* on page 13-25.

13.8.11 Understanding the register view

ARM processors support up to seven processor modes depending on the architecture version, for example User (USR), *SuperVisor Call* (SVC), and FIQ. All modes, except User mode, are referred to as *privileged* modes.

ARM processors have thirty-seven registers arranged in partially overlapping banks. There is a different register bank for each processor mode, for example USER or FIQ. Using banked registers gives rapid context switching for dealing with processor exceptions and privileged operations.

RealView Debugger displays registers in named groups to reflect how registers are banked. For example USR and FIQ for an ARM966E-S processor, shown in Figure 13-24.

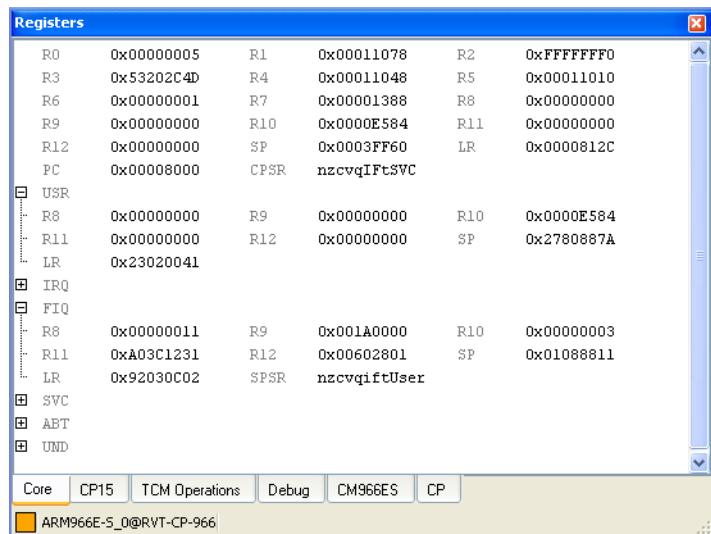


Figure 13-24 Viewing registers

At first sight, it might appear that some registers are missing or that extra registers are visible for different processor modes. For example, FIQ contains R8, R9, R10, R11, R12, SP, LR, and CPSR. These are the registers in the bank for that processor mode that are banked out when an FIQ exception occurs.

However, USR also contains R8, R9, R10, R11, R12, SP, and LR. These are banked out registers indirectly accessible with LDM and STM instructions of the form:

```
LDM rX, (r8-r14)^
STM rX, (r8-r14)^
```

These examples use the ^ suffix to specify that data is transferred into or out of the USR mode registers. The register list must not contain the PC.

You must load banked out registers from memory to modify them, and you must store them to memory to read them. However, they might be of interest when writing task context switch code or a coprocessor emulator.

Display colors in the Registers view

When using the Registers view to see register contents, RealView Debugger uses color to make the display easier to read and to highlight significant events:

- Black indicates values that are unchanged for the previous two updates
- Dark blue shows those values that have changed since the last update
- Light blue indicates a value that changed at the previous update
- Yellow indicates that the register is a read-only register
- Black -- indicates a write-only register
- Red !! indicates:
 - the address of a memory mapped register is in an area of memory that does not exist on the target
 - there has been a failure accessing the register.

See also

- the following for details of the LDM and STM instructions:
 - *ARM® Compiler toolchain Assembler Reference*
 - *ARM® Compiler toolchain Developing Software for ARM® Processors*.

13.8.12 Viewing debugger internals

Debugger internals appear in tabs in the Registers view. The tabs that appear depend on your debug target.

Viewing RVISS cycle count statistics

RVISS increments cycle count statistics during target execution.

See *Viewing statistics for RVISS targets* on page 13-74 for more details.

Viewing internal variables for connections through DSTREAM or RealView ICE

RealView Debugger uses internal variables like any other program. These are set up as part of the Debug Configuration. The variables available depend on your Debug Interface and target processor. For target connections through a DSTREAM or RealView ICE unit the debugger internals are displayed in the **Debug** tab. An example is shown in Figure 13-25.

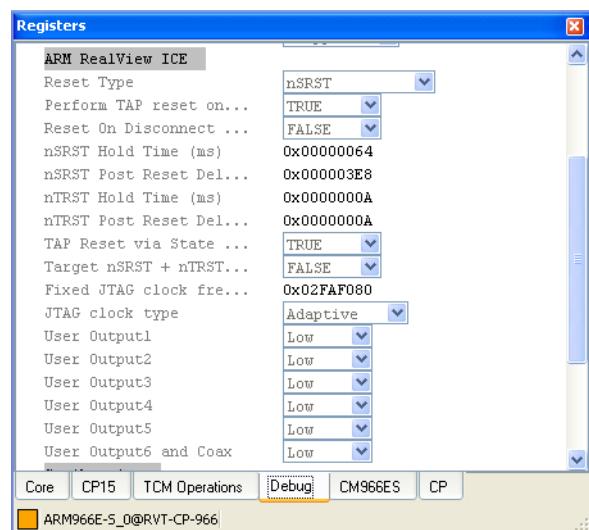


Figure 13-25 Viewing internal variables (DSTREAM or RealView ICE)

Viewing semihosting controls

Semihosting enables code running on an ARM architecture-based target to use facilities on a host computer that is running RealView Debugger. Examples of such facilities include the keyboard input, screen output, and file system I/O. The method of controlling semihosting depends on the connection you are using.

See also:

- *Viewing semihosting controls for DSTREAM or RealView ICE JTAG connections* on page 13-35
- *Viewing semihosting controls for RVISS targets* on page 13-37.

Standard semihosting behavior

When you are connected to a target through a DSTREAM or RealView ICE unit, and standard semihosting is enabled, then the target processor enters debug state when a semihosting operation is performed.

If there is ROM or FLASH at the SVC Vector, then the use of semihosting requires a hardware breakpoint on certain processor, such as an ARM7TDMI. In this case, there might be insufficient hardware breakpoint resources left to permit single instruction stepping or source level stepping, so it might be necessary to disable semihosting.

When you connect to an ARM7TDMI on an Integrator/AP board using DSTREAM or RealView ICE, the following warning messages are displayed:

- for software breakpoint modes AUTO, WATCHPOINT, and BKPT:
Warning: A software breakpoint is being used to simulate reset vector catch. This may fail to be hit if the memory is remapped when a reset occurs.
- for software breakpoint mode NONE:
Warning: Insufficient hardware resources to enable requested vector catch events. Some vector catch events have been disabled.

See also

- *Breakpoints in different memory map regions* on page 11-8
- *Specifying processor exceptions (global breakpoints)* on page 11-65
- *Viewing semihosting controls for DSTREAM or RealView ICE JTAG connections* on page 13-35
- *Viewing semihosting controls for RVISS targets* on page 13-37
- *Changing SVC numbers* on page 14-10
- *Enabling or disabling semihosting (hardware targets)* on page 14-10
- the description of RVISS cycle types in the *RealView ARMulator ISS User Guide*
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*
- the hardware documentation for the target processor you are simulating.

13.9 Viewing semihosting controls for DSTREAM or RealView ICE JTAG connections

The semihosting controls for connections through a DSTREAM or RealView ICE Debug Interface are available on the **Debug** tab in the Registers view. Standard semihosting is enabled by default.

Figure 13-26 shows an example:

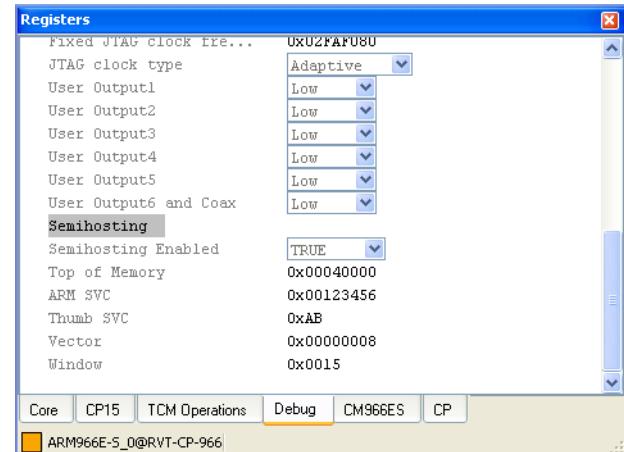


Figure 13-26 Specifying semihosting (DSTREAM or RealView ICE)

See also:

- *Changing SVC numbers* on page 14-10 if you want to change the ARM and Thumb SVC numbers.

13.9.1 The semihosting controls for DSTREAM or RealView ICE connections

The following semihosting controls for DSTREAM or RealView ICE connections are shown in Figure 13-26:

Semihosting_Enabled

Indicates the semihosting state:

True Semihosting enabled

False Semihosting disabled

Top_of_Memory

The address of top of memory.

ARM_SVC The SVC used for semihosting in ARM state.

Thumb_SVC

The SVC used for semihosting in Thumb state.

Vector The address of the semihosting vector. The default is 0x8.

Window Identifies the **StdIO** tab where any messages and prompts from the image are to be displayed.

———— Note ————

You must not change this value.

You can also use the `@semihost_symbolname` symbols to access these values. To get a list of these symbols, enter the following CLI command:

reginfo,access,match:semihost

— Note —

It is suggested that you do not modify this in the Registers view. Instead, enable or disable the processor exceptions.

See also

- *Specifying processor exceptions (global breakpoints)* on page 11-65
- *Standard semihosting behavior* on page 13-34
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the REGINFO command
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*.

13.10 Viewing semihosting controls for RVISS targets

If you connect to an RVISS model, the **Semihost** tab in the Registers view enables you to control the behavior of semihosting. Semihosting is enabled by default.

Figure 13-27 shows an example:

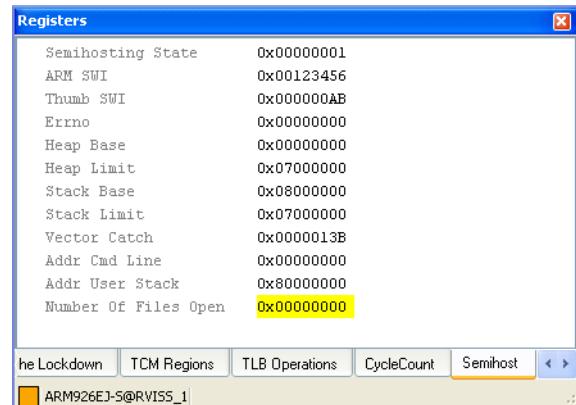


Figure 13-27 Viewing Semihost tab (RVISS)

See also:

- The semihosting controls for RVISS targets.*

13.10.1 The semihosting controls for RVISS targets

The following semihosting controls for RVISS are shown in Figure 13-27:

Semihosting_State

Indicates the semihosting state:

- 1** Semihosting enabled
- 0** Semihosting disabled

ARM_SWI The SVC used for semihosting in ARM state.

Thumb_SWI

The SVC used for semihosting in Thumb state.

Errno The errno value of the last SVC operation manipulating files, returned by the SYS_ERRNO SVC.

Heap_Base The lowest address of the system heap, returned by the SYS_HEAPINFO SVC.

Heap_Limit The highest address available for the system heap, returned by the SYS_HEAPINFO SVC.

Stack_Base The lowest address of the system stack, returned by the SYS_HEAPINFO SVC.

Stack_Limit The highest address available for the system stack, returned by the SYS_HEAPINFO SVC.

Vector_Catch

A bit mask of the current vector catch, showing the processor exceptions that are currently enabled.

Note

It is suggested that you do not modify this in the Registers view. Instead, enable or disable the processor exceptions.

Addr_Cmd_Line

The address of the command line in ARM space that is to receive the command line to the image (SWI_GetEnv).

Addr_User_Stack

The address of the user stack (SWI_GetEnv).

Number_Of_Files_Open

The number of files that the semihosting operations currently have open.

You can also use the @semihost_symbolname symbols to access these values. To get a list of these symbols, enter the following CLI command:

reginfo,access,match:semihost

See also

- *Specifying processor exceptions (global breakpoints)* on page 11-65
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the REGINFO command.
- *RealView ARMulator ISS User Guide*.

13.11 Viewing memory contents

You might want to view the contents of memory at various locations during image execution. RealView Debugger enables you to do this in many ways.

See also:

- *Toggling the display of the Memory view toolbar*
- *Examining memory in the Memory view*
- *Viewing memory at a specific address* on page 13-40
- *Viewing the memory at a register value* on page 13-41
- *Viewing memory associated with a variable in source code* on page 13-41
- *Viewing memory at a variable value by copying and pasting* on page 13-42
- *Viewing memory at the destination of a branch instruction* on page 13-42
- *Viewing memory at a variable in the Locals view* on page 13-43
- *Viewing memory at a watch variable* on page 13-43
- *Viewing memory at a previous start address* on page 13-44
- *Viewing memory for multiple processors* on page 13-44
- *Considerations when performing the view memory operations* on page 13-45.

13.11.1 Toggling the display of the Memory view toolbar

The Memory view includes a toolbar that enables you to change the location and formatting of the memory view. Figure 13-28 shows an example:

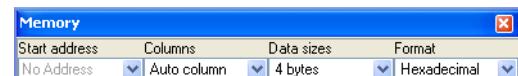


Figure 13-28 Memory view toolbar

This toolbar is displayed by default.

To toggle the display of the Memory view toolbar:

1. Right-click on the contents of a memory location to display the context menu.
2. Select **Toolbar** from the context menu. This toggles the display of the toolbar.

To toggle the display of the toolbar labels, select **Toolbar Label** from the context menu.

If the Memory view toolbar is currently hidden, you can temporarily display the toolbar if you select **Set Start Address** from the Memory view context menu. The toolbar remains visible until you enter a new start address.

13.11.2 Examining memory in the Memory view

To examine the contents of memory in the Memory view:

1. Connect to your target.
2. Load your image, for example, dhystone.axf.
3. Select **Memory** from the **View** menu to display the Memory view as a floating view. Figure 13-29 on page 13-40 shows an example. No memory locations are displayed when you first connect to a target.

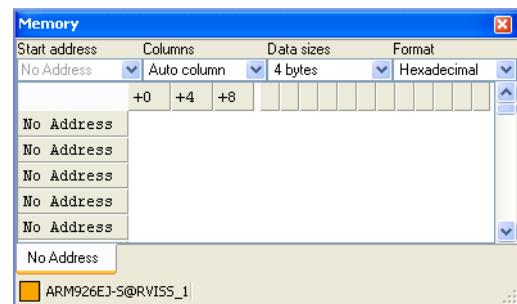


Figure 13-29 Memory view

4. Enter the required location in the Start address field of the Memory view toolbar. The tab name changes to the specified start address.

————— Note —————

The Memory view toolbar is displayed by default.

You can specify the start address as:

- an address in hexadecimal, for example 0x000008A4
- a C or C++ expression that RealView Debugger computes to obtain the starting address, for example Int_Glob+64
- a macro that returns an address, for example thisaddr()
- any valid expression using constants and symbols.

You can also use the drop-down arrow to select the start address from values you have entered previously.

The Memory view is updated. Figure 13-30 shows an example:

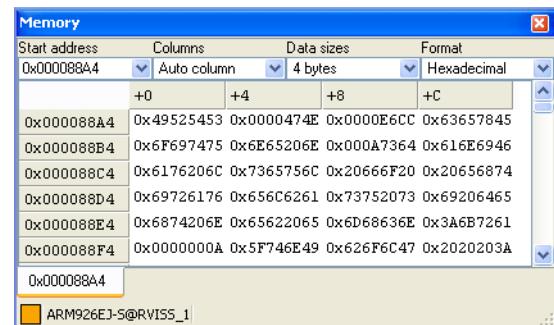


Figure 13-30 Memory view showing address contents

See also

- [Connecting to a target](#) on page 3-27
- [Loading an executable image](#) on page 4-4
- [Toggling the display of the Memory view toolbar](#) on page 13-39
- [Display colors in the Memory view](#) on page 13-56.

13.11.3 Viewing memory at a specific address

To display memory at a specific address, enter the address of the first memory location to be viewed in the Start Address field. The address can be:

- An explicit value.

- The result of a C or C++ expression.
- A return value from a macro.
- The value of a register or variable. For example, to display memory at the address pointed to by the SP register, enter @SP.

The Memory view is updated to show the memory starting at the address you specified.

13.11.4 Viewing the memory at a register value

To view the memory at a register value:

1. Display the Registers view if it is not already visible.
2. Right-click on the register containing the address where you want to view memory, for example, the PC register. A context menu is displayed.
3. Select **View Memory at Value** from the context menu to display the Memory view. The start address is set to the value of the chosen register. Figure 13-31 shows an example:

The screenshot shows a Windows-style application window titled "Memory". At the top, there are dropdown menus for "Start address" (set to 0x00008000), "Columns" (set to "Auto column"), "Data sizes" (set to "1 byte"), and "Format" (set to "Hexadecimal"). The main area is a grid of memory data. The first few rows of data are as follows:

	+0	+1	+2	+3	+4	+5	+6	+7	+8
0x00008000	0x00	0x00	0x00	0xEB	0xB0	0x07	0x00	0xEB	0x2C
0x00008009	0x00	0x8F	0xE2	0x00	0x0C	0x90	0xE8	0x00	0xA0
0x00008012	0x8A	0xE0	0x00	0xB0	0x8B	0xE0	0x01	0x70	0x4A
0x0000801B	0xE2	0x0B	0x00	0x5A	0xE1	0x00	0x00	0x00	0x1A
0x00008024	0xA8	0x07	0x00	0xEB	0xF0	0x00	0xBA	0xE8	0x18
0x0000802D	0xE0	0x4F	0xE2	0x01	0x00	0x13	0xE3	0x03	0xF0
0x00008036	0x47	0x10	0x03	0xF0	0xA0	0xE1	0x38	0x65	0x00
0x00008000	0x00								

Figure 13-31 Memory view at an address in a register

See also

- *Examining registers in the Registers view* on page 13-25
- *Considerations when performing the view memory operations* on page 13-45.

13.11.5 Viewing memory associated with a variable in source code

If a variable is in the current scope, then you can view the memory at the value or address of that variable.

To view the memory associated with a variable in your source code:

1. Open the source file containing the variable, if it is not already open.
2. Locate the variable in the source file tab.
3. Right-click on the variable name to display the context menu.
4. Do one of the following:
 - Select **View Memory at Value** from the context menu to display the Memory view. The start address is set to the value of the chosen variable. Figure 13-32 on page 13-42 shows an example:

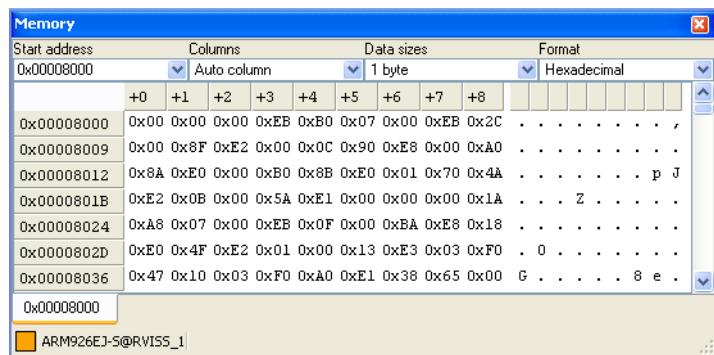


Figure 13-32 Memory view at a value in a variable

- Select **View Memory at Address** from the context menu to display the Memory view. The start address is set to the address of the chosen variable.

Note

This option cannot be used for register and constant variables.

See also

- Considerations when performing the view memory operations* on page 13-45.

13.11.6 Viewing memory at a variable value by copying and pasting

To view memory at the value of that variable by copying and pasting:

- Double-click on the variable name in your source to select the variable.
The variable must be in scope.
- Right-click on the selected variable to display the context menu.
- Select **Copy** from the context menu.
- Click on the Start Address field of the Memory view.
- Remove any text in the Start Address field.
- Press Ctrl+V to paste the variable name in the Start Address field.
The value of the variable is used as the address. To use the address of the variable, insert & before the variable name.
- Press Enter to update the contents of the Memory view.

See also

- Copying and pasting variables from the Locals view to the Watch view* on page 13-18.

13.11.7 Viewing memory at the destination of a branch instruction

To view memory at the destination of a branch instruction:

- Click the **Disassembly** tab in the Code window.
- Locate the branch instruction of interest.
- Right-click on the destination to display the context menu.

4. Do one of the following:
 - Select **View Memory at Value** from the context menu to display the Memory view. The start address is set to the destination address of a branch instruction.
 - Select **View Memory at Address** from the context menu.

See also

- *Considerations when performing the view memory operations* on page 13-45.

13.11.8 Viewing memory at a variable in the Locals view

To view memory at a variable in the Locals view:

1. Display the Locals view.
2. Locate the required variable.

Note

The variable must be in scope.

3. Right-click on the variable name to display the context menu.
4. Either:
 - Select **View Memory at Value** from the context menu to display the memory using the value of the selected as the start address.
 - Select **View Memory at Address** from the context menu to display the memory using the address of the selected variable as the start address.

Note

This option cannot be used for register and constant variables.

See also

- *Viewing variables for the current context* on page 13-14
- *Considerations when performing the view memory operations* on page 13-45.

13.11.9 Viewing memory at a watch variable

To view memory at a watch variable:

1. Display the Watch view.
2. Locate the required watch variable.

Note

The variable must be in scope.

3. Right-click on the watch variable name to display the context menu.
4. Either:
 - Select **View Memory at Value** from the context menu to display the memory using the value of the watch variable as the start address.
 - Select **View Memory at Address** from the context menu to display the memory using the address of the watch variable as the start address.

Note

This option cannot be used for register and constant variables.

See also

- *Examining watches in the Watch view* on page 13-70
- *Considerations when performing the view memory operations* on page 13-45.

13.11.10 Viewing memory at a previous start address

Each time you enter an address in the Start Address field, the address is stored on the history list. This list is available as a drop-down list for the Start Address field.

To view the memory at a previous start address:

1. Display the Memory view toolbar, if it is not already visible.

Note

The Memory view toolbar is displayed by default.

2. Select a previous address from the Start Address drop-down list in the Memory view toolbar.

The Memory view changes to show the memory at the selected start address.

3. Repeat this operation as required.

See also

- *Toggling the display of the Memory view toolbar* on page 13-39.

13.11.11 Viewing memory for multiple processors

If you are debugging multiple processors you can access different memory views on processor at the same time. To do this:

1. Connect to each processor of your multiprocessor system.
 2. Display multiple Code windows.
 3. Attach each Code window to a different processor.
 4. In each Code window, display the Memory view for the associated processor.
- RealView Debugger enables you to set up several Memory views with different formatting options for each.

Viewing shared memory

If one processor makes changes to a region of memory that is shared by another processor, then you might have to update the Memory view for the other processor at frequent intervals to see those changes.

See also

- *Connecting to multiple targets* on page 3-46
- *Displaying multiple Code windows* on page 7-8
- *Attaching a Code window to a connection* on page 7-10

- *Sharing resources between multiple targets* on page 7-28
- *Toggling the display of the Memory view toolbar* on page 13-39.
- *Manually updating values in the Memory view* on page 13-54
- *Performing timed updates for RealMonitor and OS-aware targets* on page 13-54.

13.11.12 Considerations when performing the view memory operations

The first time you perform the **View Memory At Address** or **View Memory At Value** operations in a debugging session, a new instance of the Memory view is displayed as a floating view. Subsequent use of these operations in the same Code window use the same instance of the Memory view to display the memory contents.

13.12 MMU page tables views

If your ARM architecture-based processor supports a *Memory Management Unit* (MMU), you can view the two-level page tables that are stored in main memory. Two options are available, depending on your MMU architecture:

- page tables for ARM architecture v4 (ARMv4) and ARMv5 architecture
- page tables for ARMv6 and ARMv7 architectures.

See also:

- *Viewing the MMU page tables*
- *Switching between the Level 1 and Level 2 descriptors* on page 13-48
- *Walking through an MMU page table* on page 13-49.

13.12.1 Viewing the MMU page tables

To view the MMU page tables:

1. Select **User Memory Formats** from the **View** menu.
2. From the sub-menu, select the required option for your MMU architecture:
 - For ARMv4 and ARMv5 architectures:
 - **Pagetables Arch v4/v5 → Level1 Descriptors**
 - **Pagetables Arch v4/v5 → Level2 Descriptors**
 - For ARMv6 and ARMv7 architectures:
 - **Pagetables Arch v6/v7 → Level1 Descriptors**
 - **Pagetables Arch v6/v7 → Level2 Descriptors**.

Note

Make sure you use the correct option for your MMU architecture.

A new floating Memory view is displayed showing the selected pagetable descriptors. Figure 13-33 shows an example:

Start address	Columns	Display sizes	Format
S:0x77EFC000	Auto column	4 bytes	User format...
+0			
S:0x77EFC000	L1_LEVEL2_TABLEBASE2000A701	80029_8_S	
S:0x77EFC004		FAULT	
S:0x77EFC008		FAULT	
S:0x77EFC00C		FAULT	
S:0x77EFC010		FAULT	
S:0x77EFC014		FAULT	
S:0x77EFC018		FAULT	
S:0x77EFC01C	L1_LEVEL2_TABLEBASE20A04805	82812_0_S	
S:0x77EFC020	L1_SECTION	A_0_S_ng_s_xn_00_0_010	
S:0x77EFC024		--	
S:0x77EFC028		FAULT	
S:0x77EFC02C		FAULT	
S:0x77EFC030	L1_SECTION	2001802_20_S_ng_s_xn_02_0_100	
S:0x77EFC034	L1_LEVEL2_TABLEBASE49014009124050_0_NS		
S:0x77EFC038	L1_SECTION	49021802490_S_NG_s_xn_02_0_100	
S:0x77EFC000			
	ARM1176JZF-S_0@RVI_1		

Figure 13-33 ARMv6 and ARMv7 MMU table (Level 1 descriptors)

3. Change Columns to **1**. This prevents the view being shown in multiple columns, if you expand the view to the right. A single-column view is less confusing.
4. To view the details of an MMU table entry, double-click on that entry. The Level 1 descriptors dialog box or Level 2 descriptors dialog box is displayed as appropriate. For the example in Figure 13-33 on page 13-46, double-click on the entry at **0x77EFC030**. The Level 1 descriptors dialog box shown in Figure 13-34 is displayed.

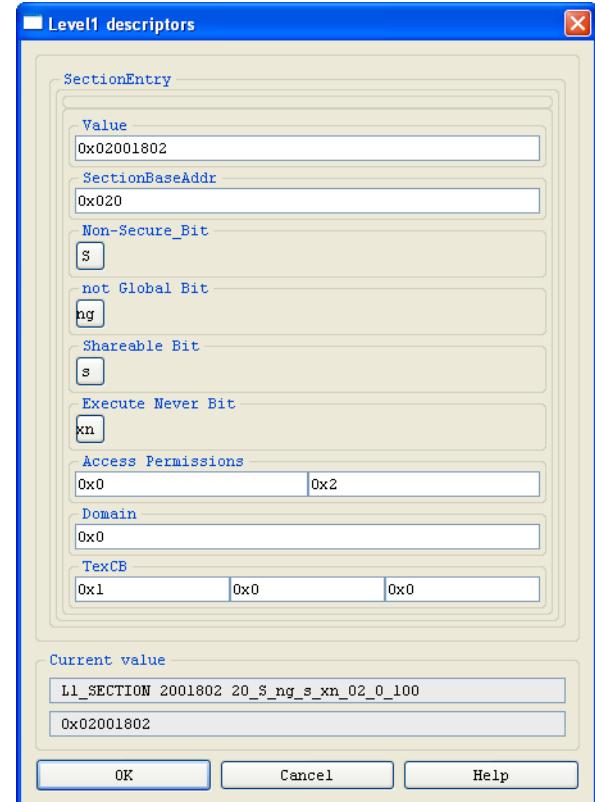


Figure 13-34 Level 1 descriptors dialog box

See also

- *Switching between the Level 1 and Level 2 descriptors* on page 13-48
- *Walking through an MMU page table* on page 13-49
- *Adding memory views for different locations* on page 13-55
- *Moving a memory view to another Memory view* on page 13-56.

13.12.2 Switching between the Level 1 and Level 2 descriptors

If you have an existing MMU table view, you can switch between the Level 1 and Level 2 descriptors of that MMU. An example MMU table view is shown in Figure 13-33 on page 13-46.

To switch between the Level 1 and Level 2 descriptors:

1. In the Memory view that shows an existing MMU table, right-click on the value of a memory location to display the context menu.
2. From the **Format** sub-menu, select the required MMU table view:
 - For ARMv4 and ARMv5 architectures:
 - **Pagetables Arch v4/v5 → Level1 Descriptors**
 - **Pagetables Arch v4/v5 → Level2 Descriptors**
 - For ARMv6 and ARMv7 architectures:
 - **Pagetables Arch v6/v7 → Level1 Descriptors**
 - **Pagetables Arch v6/v7 → Level2 Descriptors.**

————— **Note** —————

The sub-menu does not include an option for the descriptors that are currently being viewed.

————— **Note** —————

Make sure you use the correct options for your MMU architecture.

See also

- *Viewing the MMU page tables* on page 13-46
- *Walking through an MMU page table* on page 13-49
- *Adding memory views for different locations* on page 13-55
- *Moving a memory view to another Memory view* on page 13-56.

13.12.3 Walking through an MMU page table

To walk through the MMU page table:

1. Right-click on a page table entry in the Memory view.
2. Select **View Memory at Address** from the context menu. The Memory view changes to show the destination page table entry.

As you walk through each page table entry, the entry you selected is stored in a history list. To browse this history list:

1. Right-click on a page table entry.
2. Select either **Back** or **Forward** from the context menu as required.

See also

- *Viewing the MMU page tables* on page 13-46
- *Switching between the Level 1 and Level 2 descriptors* on page 13-48
- *Adding memory views for different locations* on page 13-55
- *Moving a memory view to another Memory view* on page 13-56.

13.13 Managing the display of memory in the Memory view

The following sections describe how to manage how memory contents in the Memory view:

- *Changing the display size for viewing memory contents*
- *Formatting the display of memory contents* on page 13-51
- *Displaying the ASCII view in separate columns* on page 13-52
- *Changing the number of columns displayed* on page 13-52
- *Coloring memory contents* on page 13-53
- *Toggling automatic updates of the Memory view* on page 13-53
- *Manually updating values in the Memory view* on page 13-54
- *Performing timed updates for RealMonitor and OS-aware targets* on page 13-54
- *Adding memory views for different locations* on page 13-55
- *Deleting a memory view* on page 13-56
- *Moving a memory view to another Memory view* on page 13-56
- *Display colors in the Memory view* on page 13-56.

13.13.1 Changing the display size for viewing memory contents

The display size used for viewing memory contents varies depending on the data types supported by your target processor.

To change the display size for viewing the memory contents, select the required size from the Size of fields list in the Memory view toolbar. Figure 13-35 shows an example Memory view displaying two byte values.

Memory						
Start address	Columns	Data sizes				
0x00008000	Auto column	2 bytes				
	+0	+2	+4	+6	+8	+A
0x00008000	0x0000	0xEB00	0x07B0	0xE000	0x002C	0xE28F .
0x0000800C	0x0C00	0xE890	0xA000	0xE08A	0xB000	0xE08B .
0x00008018	0x7001	0xE24A	0x000B	0xE15A	0x0000	0x1A00 .

Figure 13-35 Size of fields list in Memory view

The options available are:

- | | |
|----------------|---|
| Default | Displays memory contents in the format specified as the minimum memory access size for the target. This is the default. |
| 1 byte | Each column displays eight bits of data. |
| 2 bytes | Each column displays 16 bits of data. If your debug target is an ARM architecture-based processor, the memory contents are aligned on two-byte boundaries. |
| 4 bytes | Each column displays 32 bits of data. If your debug target is an ARM architecture-based processor, the memory contents are aligned on four-byte boundaries. |
| 8 bytes | Each column displays 64 bits of data. |

13.13.2 Formatting the display of memory contents

The display format used for viewing memory contents varies depending on the data types supported by your target processor.

To change the display format for viewing the memory contents, select the required format from the Format of fields list in the Memory view toolbar. The options available are:

ASCII Displays the memory values in ASCII format.

The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.).

Binary Displays memory values in binary format including leading zeros.

Decimal Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

Disassembly

Displays the memory contents as disassembled code. The disassembly is displayed in a single column. The Number of columns list and the Size of fields list in the toolbar are disabled. Also, if you have the ASCII view displayed, it is hidden until you select another format.

To change the disassembly format:

1. Right-click on a line of disassembly in the DISASSEMBLY column to display the context menu.
2. Select the required format from the **Disassembly Format** submenu. The options available are:
 - **Auto (ARM, Thumb, Bytecode, Literals)**
 - **ARM**
 - **Thumb**
 - **Bytecode**
 - **Thumb-2EE**.

Floating point

Displays memory values in floating point format, depending on the currently selected size:

4 bytes Displays values in floating point IEEE format, occupying four bytes, for example:

2.5579302e-041

8 bytes Displays values in floating point IEEE format, occupying eight bytes, for example:

4.71983561663e+164

Any other size

This format cannot be displayed, and the following error is displayed in each memory location:

Error: The requested format is not available

Change the size to four or eight bytes as required.

Hexadecimal

Displays memory values in hexadecimal format including leading zeros, together with the 0x prefix.

This is the default display format.

Octal	Displays memory values in octal format including leading zeros, together with the o prefix.
--------------	--

Figure 13-36 shows an example Memory view displaying disassembly format.

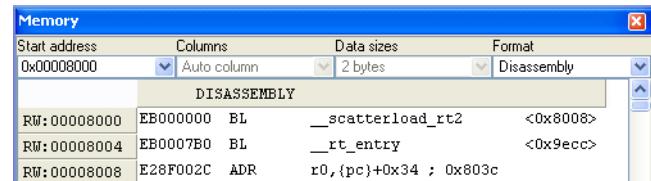


Figure 13-36 Format of fields list in the Memory view

See also

- *Changing the display size for viewing memory contents* on page 13-50
- *Displaying the ASCII view in separate columns*.

13.13.3 Displaying the ASCII view in separate columns

You can display the ASCII view of memory locations in separate columns, and still have the memory values visible in your chosen format.

To display the ASCII view of memory locations in separate columns:

1. Right-click on the contents of a memory location in the Memory view to display the context menu.
2. Select **Show ASCII** from the context menu. Additional columns showing the equivalent ASCII characters are displayed to the right of the main memory columns.

If you select **Disassembly** from the Format of fields list in the Memory view toolbar, then the ASCII view is hidden until you select another format.

13.13.4 Changing the number of columns displayed

When the Memory view first opens, the number of columns you can see is fixed at four columns. To change the number of columns visible in the display:

1. Display the Memory view toolbar, if it is not already visible.

————— Note —————

The Memory view toolbar is displayed by default.

2. Select the required number of columns to display from the Number of columns list in the Memory view toolbar. Figure 13-37 shows an example:

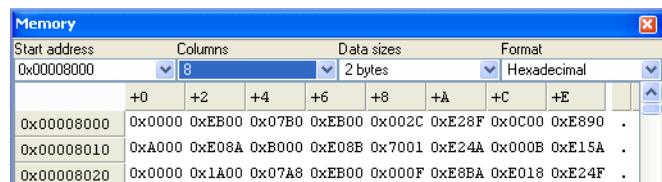


Figure 13-37 Number of columns list in Memory view

This number excludes the columns used for displaying the ASCII view.

You can explicitly choose to display from one column up to 16 columns. However, if you select **Auto column**, then the maximum number of columns displayed depends on the width of the Memory view, and whether or not the ASCII view is displayed.

See also

- *Toggling the display of the Memory view toolbar* on page 13-39
- *Displaying the ASCII view in separate columns* on page 13-52.

13.13.5 Coloring memory contents

By default, the contents of memory are colored with black text on a white background. However, RealView Debugger can color the memory contents in the Memory view based on the custom memory mapping or caching. To choose the required coloring of the memory contents:

1. Select **Memory** from the **View** menu to display the Memory view.
2. Enter an address in the Start address field of the toolbar.
3. Right-click on a memory cell to display the context menu.
4. Select one of the following options on the **Coloring** submenu:
 - **Memory Map** to color the memory contents based on the custom memory map defined in the Debug Configuration. This is the default.
 - **Cache** to color the memory contents of cached memory areas, if supported by the currently connected target.

See also

- *Display colors in the Memory view* on page 13-56
- *Chapter 9 Mapping Target Memory*.

13.13.6 Toggling automatic updates of the Memory view

By default, the Memory view updates automatically. To toggle the automatic update of the Memory view:

1. Right-click in the Memory view to display the context menu.
2. Select **Automatic Update** from the context menu.

Automatic updating refreshes the Memory view as soon as execution stops at a point in your image, such as at a breakpoint.

Be aware that for shared memory regions in a multiprocessor development platform, changes to the memory region are updated only for the memory view of the target that makes the change. If you have other Code windows open, then to see the changes in each Code window you must manually update the memory view in those Code windows.

Freezing the contents of the Memory view

To freeze the contents of the Memory view, disable automatic updating. You can manually update the view contents if required.

Freezing the contents of the Memory view enables you to compare the contents with those in a second Memory view.

See also

- *Sharing resources between multiple targets* on page 7-28
- *Updating windows and views when a breakpoint activates* on page 12-4
- *Manually updating values in the Memory view.*

13.13.7 Manually updating values in the Memory view

To manually update the values in the Memory view:

1. Disable automatic updates.
2. Right-click on the contents of a memory location to display the context menu.
3. Select **Update View** from the context menu. The memory contents are updated.

Note

You can also update the Memory view when a breakpoint activates.

See also

- *Sharing resources between multiple targets* on page 7-28
- *Updating windows and views when a breakpoint activates* on page 12-4
- *Toggling automatic updates of the Memory view* on page 13-53.

13.13.8 Performing timed updates for RealMonitor and OS-aware targets

If you are using RealMonitor or an OS extension, the memory display can be updated at a specified time interval during program execution.

To perform a timed update of the Memory view:

1. Connect to a target that is configured to run RealMonitor or that is running an OS-aware application.
2. Right-click on the contents of a memory location to display the context menu.
3. Select **Timed Update Period...** from the context menu to display the Timed Update Period dialog box. Figure 13-38 shows an example:

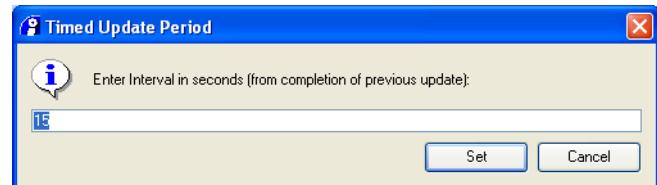


Figure 13-38 Timed Update Period dialog box

4. Enter the interval, in seconds, between window updates. The default is 15 seconds. This value is used only when **Timed Update** is enabled.
5. Click **OK** to set the interval.
6. Right-click on the contents of a memory location to display the context menu.
7. Select **Timed Update** from the context menu. This option is available when:
 - RealMonitor is activated

- the target is in *Running System Debug* (RSD) mode and where supported by the underlying debug target.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43
- RealView Debugger RTOS Guide*.

13.13.9 Adding memory views for different locations

In the Memory view, you can add views to multiple memory regions:

- Display the Memory view if it is not already visible.
- Enter the required location in the Start address field of the Memory view toolbar.

For example, enter 0x8000. The tab name changes to the specified start address, shown in Figure 13-39.

The screenshot shows a Windows-style window titled "Memory". The toolbar at the top has dropdown menus for "Start address" (set to 0x000008000), "Columns" (Auto column), "Data sizes" (4 bytes), and "Format" (Hexadecimal). Below the toolbar is a table with columns for Address, Data, and Format. The first row shows the header: "Start address" (0x000008000), "Columns" (Auto column), "Data sizes" (4 bytes), and "Format" (Hexadecimal). The table contains several rows of memory data, starting from 0x000008000 up to 0x000008060. The last row is highlighted in yellow. At the bottom of the window, there is a status bar with the text "ARM926EJ-S@RVISS_1".

Figure 13-39 Memory view showing address contents

- Right-click on the tab for the memory region to display the context menu.
- Select **Duplicate View** from the context menu. A new tab is created with the same start address as the chosen tab.
- Enter the new location in the Start address field of the Memory view toolbar.

For example, enter 0x9000. The tab name changes to the specified start address, shown in Figure 13-40.

The screenshot shows a Windows-style window titled "Memory". The toolbar at the top has dropdown menus for "Start address" (0x000009000), "Columns" (Auto column), "Data sizes" (4 bytes), and "Format" (Hexadecimal). Below the toolbar is a table with columns for Address, Data, and Format. The first row shows the header: "Start address" (0x000009000), "Columns" (Auto column), "Data sizes" (4 bytes), and "Format" (Hexadecimal). The table contains several rows of memory data, starting from 0x000009000 up to 0x000009060. The last row is highlighted in yellow. At the bottom of the window, there is a status bar with the text "ARM926EJ-S@RVISS_1".

Figure 13-40 Memory view with multiple memory views

13.13.10 Deleting a memory view

To delete a memory view from a Memory view:

1. Right-click on the tab of the memory view to be deleted to display the context menu.
2. Select **Delete View** from the context menu. The tab for the memory view is deleted.

————— **Note** —————

If you perform this operation in a Memory view with a single memory view, then the Memory view closes.

13.13.11 Moving a memory view to another Memory view

To move a memory view in one Memory view to another Memory view, click and drag the tab of the required memory view as follows:

- If you drop the selected memory view on the tab control of another Memory view, then the tab is added to that Memory view.
- If you drop the selected memory view at a point that is not on the tab control of another Memory view, then a new Memory view is created containing the tab.

13.13.12 Display colors in the Memory view

When using the Memory view to see memory contents, RealView Debugger uses color to make the display easier to read and to highlight significant events. The colors used depend on the type of coloring you have selected, and on whether or not memory mapping is enabled.

Colors used when memory mapping is enabled

When memory mapping is enabled, the following colors are used for memory contents:

- Black text on a white background specifies RAM or memory that can be modified.
- Blue text shows those contents that have changed since the last update. Light blue text indicates a previous update.
- Black text on a yellow background indicates the contents of ROM.
- Green text on a white background indicates Flash memory known to RealView Debugger.

————— **Note** —————

If the Flash memory locations are shown with a yellow background in the Memory view, then the `Flash_type` setting for the Flash memory area in the BCD file is either pointing to an invalid FME file, or is not set.

- Red ** indicates:
 - no memory is currently defined in the memory map at this location
 - memory at this location is defined as Auto meaning it is determined when loading your application program
 - memory is defined as *prompt* meaning that you are prompted to confirm the usage when loading the application.

- Red !! indicates:
 - a memory map block is defined for an area of memory that does not exist on the target
 - a failure in performing the memory operation.

Colors used when memory mapping is disabled

When memory mapping is disabled, the following colors are used for memory contents:

- Black text on a white background for all existing memory areas.
- Blue text shows those contents that have changed since the last update. Light blue text indicates a previous update.
- Red !! indicates:
 - non-existent memory
 - a failure in performing the memory operation.

Colors used for caching

Table 13-1 summarizes the colors used for cached memory.

Table 13-1 Colors used for cached memory contents

Cache	Background color
L1 cache (clean)	Light Green
L1 cache (dirty)	Red
L2 cache (clean)	Yellow
L2 cache (dirty)	Brown

For all other existing memory areas, the following colors are used for the memory contents:

- Black text on a white background for all existing memory areas.
- Blue text shows those contents that have changed since the last update. Light blue text indicates a previous update.

The contents of non-existent memory depends on whether memory mapping is enabled or disabled.

See also

- *Editing a memory map entry* on page 9-20
- *Viewing memory contents* on page 13-39.

13.14 Viewing the Stack

The stack holds function return information and local variables. You can examine the stack when image execution stops at a specific point.

See also:

- *Before you start*
- *Examining the Stack*
- *Viewing the Stack at a specified address* on page 13-59
- *Viewing the Stack at an address contained in a selected Stack entry* on page 13-60
- *Viewing the symbol at a Stack address* on page 13-60
- *Viewing the symbol at an address in a selected Stack entry* on page 13-61
- *Viewing the Stack by re-evaluating a C/C++ expression* on page 13-61
- *Copying the contents of the Stack* on page 13-61
- *Formatting the Stack contents* on page 13-62
- *Changing the data size for Stack contents* on page 13-62.

13.14.1 Before you start

Before you can examine the Stack:

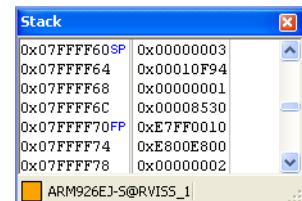
1. Connect to your target.
2. Load the required image. For example, dhystone.axf.
3. Run the image to a specific point.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- *Chapter 8 Executing Images*.

13.14.2 Examining the Stack

To examine the Stack, select **Stack** from the **View** menu. The Stack view is displayed as a floating view, shown in Figure 13-41.



A screenshot of a floating window titled "Stack". The window contains a table with two columns of memory addresses and their corresponding values. The first column is labeled "Address" and the second is "Value". The addresses listed are 0x07FFFF60, 0x07FFFF64, 0x07FFFF68, 0x07FFFF6C, 0x07FFFF70, 0x07FFFF74, and 0x07FFFF78. The values are 0x00000003, 0x00010F94, 0x00000001, 0x000008530, 0xE7FF0010, 0xE800E800, and 0x00000002 respectively. The bottom row of the table has a yellow background and contains the text "ARM926EJ-S@RVISS_1".

Figure 13-41 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.

The stack is displayed in columns:

Address The left column contains the memory addresses of the stack.

In some target processors that use a Harvard architecture, the address is prefixed with **D:** to show that this is a data address. You must include this prefix when specifying such an address as the starting address, as follows:

D:*address*

- Value** The right column displays the contents of the addresses in the stack.

As with the Memory view, the memory display in the Stack view is color-coded for easy viewing and to enable you to monitor changes.

Demonstration

The following procedure uses the example Dhystone image to demonstrate the use of the Stack view:

1. Click the **Locate PC** button on the Debug toolbar to view the source file dhry_1.c.
2. Set a default breakpoint by double-clicking on line 301.
3. Click **Run** to start execution.
4. Enter 5000 when asked for the number of runs. The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
5. Monitor changes in the Stack view as you step through your program, for example by clicking **Step** on the Debug toolbar.
6. Double-click on the red marker disc to clear the breakpoint at line 301.

13.14.3 Viewing the Stack at a specified address

To view the Stack at a specified address:

1. Right-click on a stack address (in the left column) to display the context menu.
2. Select **Set Start Address...** from the context menu to display the start address dialog box. Figure 13-42 shows an example. If you have set a start address previously, this last expression you used is displayed in the dialog box.



Figure 13-42 Start address dialog box

3. Enter an explicit address, an expression, or a register (for example @R9) as the new address.
- If you leave the expression field blank, or remove an existing expression, in the address prompt dialog box, RealView Debugger reverts to using the default stack pointer register. That is, R13 (see Figure 13-18 on page 13-26).
4. Click **Set** to confirm your choice and close the address prompt box. The new start address is marked by *Expression Pointer* (EP), located at the bottom of the stack, shown in Figure 13-43 on page 13-60.

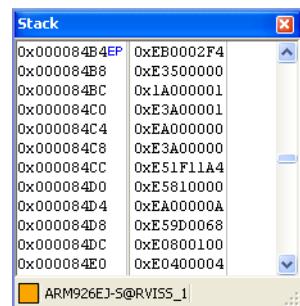


Figure 13-43 Expression Pointer in the Stack view

Note

The SP register is unchanged, and the target is unaffected.

You can also use the drop-down arrow to select an expression from a browser or to re-use a value entered previously. The drop-down also gives access to your list of personal favorites where you can store a memory address for re-use in this, or future, debugging sessions.

13.14.4 Viewing the Stack at an address contained in a selected Stack entry

To view the Stack at an address contained in a selected Stack entry:

1. Right-click on the contents of the required Stack entry to display the context menu.
2. Select **Set Start Address from Content** from the context menu.

The new start address is marked by *Expression Pointer* (EP), located at the bottom of the stack, shown in Figure 13-43.

Note

The SP register is unchanged, and the target is unaffected.

13.14.5 Viewing the symbol at a Stack address

To view the symbol at a Stack address:

1. Right-click on the contents of the required Stack entry to display the context menu.
2. Select **Display Symbol at Address** from the context menu to display the Prompt dialog box. This shows the symbol, if any, at the address of the chosen Stack entry. Figure 13-44 shows an example. If no symbol is present at the address, the text <None> is shown in place of the symbol in the dialog box.

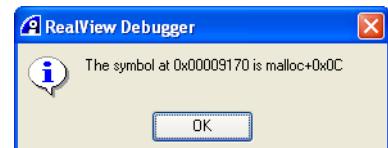


Figure 13-44 Symbol at a Stack address

3. Click **OK** to close the dialog box.

13.14.6 Viewing the symbol at an address in a selected Stack entry

To view the Stack at a previous start address:

1. Right-click on the contents of the required Stack entry to display the context menu.
2. Select **Display Symbol from Content** from the context menu to display the Prompt dialog box. This shows the symbol, if any, at the address in the chosen Stack entry. Figure 13-45 shows an example. If no symbol is present at the address, the text <None> is shown in place of the symbol in the dialog box.



Figure 13-45 Symbol at address in a selected Stack entry

3. Click **OK** to close the dialog box.

13.14.7 Viewing the Stack by re-evaluating a C/C++ expression

Where you have used a C/C++ expression to compute the start address, then you can re-evaluate that expression.

To view the Stack at the start address by re-evaluating the C/C++ expression that you entered:

1. Right-click in the address column of the Stack view to view the context menu.
2. Select **Re-evaluate Start Address** from the context menu.

The Stack view changes to show the Stack at the new start address obtained after re-evaluating the expression. The new start address is marked by *Expression Pointer* (EP), located at the bottom of the stack, shown in Figure 13-43 on page 13-60.

————— **Note** —————

The SP register is unchanged, and the target is unaffected.

13.14.8 Copying the contents of the Stack

To copy the contents of the Stack:

1. Left-click on the first Stack address in the range to be copied, and drag the cursor to the last Stack address in the range.
2. Right-click on a Stack entry in the selected range to display the context menu.
3. Select **Copy** from the context menu.
4. Paste the copied stack contents into the application of your choice, such as a text editor.

13.14.9 Formatting the Stack contents

To change the way stack contents are displayed:

1. Right-click on any Stack entry to display the context menu.
2. Select the required format from the context menu:

Signed Decimal

Displays the memory contents as negative or positive values where the maximum absolute value is half the maximum unsigned decimal value.

Unsigned Decimal

Displays the memory contents from 0 up to the highest value that can be stored in the number of bits available.

Hexadecimal

Displays memory contents as hexadecimal numbers.

Hexadecimal with leading Zeros

Displays memory values in hexadecimal format including leading zeros.

This is the default display format for data values in this view.

Show ASCII

Adds another column to the Stack view, on the right hand side, to show the ASCII value of the memory contents.

ASCII format displays column values as characters. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Any non-printable value is represented by a period (.).

13.14.10 Changing the data size for Stack contents

The display format used for viewing memory contents varies depending on the data types supported by your target processor.

To change the display format used for viewing Stack contents:

1. Right-click on any Stack entry to display the context menu.
2. Select the required data size from the context menu:

Default Each column displays 32 bits of data. Where your debug target is an ARM processor words are aligned on 4-byte boundaries.

Bytes Each column displays 8 bits of data.

Half Words

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

Words Each column displays 32 bits of data. Where your debug target is an ARM processor words are aligned on 4-byte boundaries.

Double Words

Each column displays 64 bits of data.

13.15 Viewing the Call Stack

The Call Stack enables you to follow the flow of your application through the hierarchical structure. It also enables you to traceback through your code.

See also:

- *Examining the Call Stack*
- *Copying a Call Stack entry on page 13-64*
- *Toggling automatic updates of the Call Stack view on page 13-64*
- *Manually updating the Call Stack view on page 13-65*
- *Displaying the properties of a Call Stack entry on page 13-65*
- *Operations you can perform on the Call Stack on page 13-66*.

13.15.1 Examining the Call Stack

To examine the Call Stack:

1. Select **Call Stack** from the **View** menu to display the Call Stack view. Figure 13-46 shows an example:

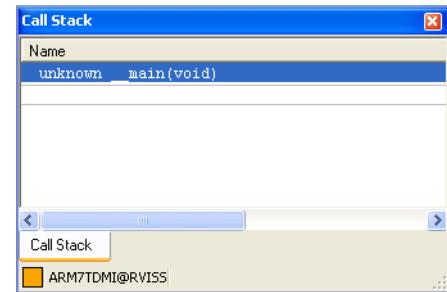


Figure 13-46 Call Stack view

Multistatement lines in the Call Stack

If an entry is part of a multistatement line, the Call Stack view shows the information in the form of line and column details. Figure 13-47 shows an example:

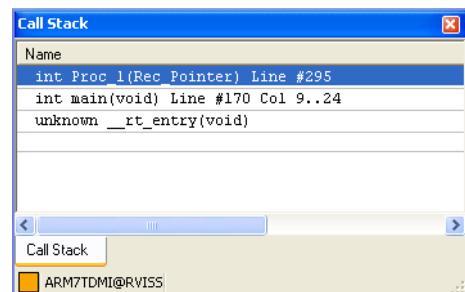


Figure 13-47 Multistatement details in the Call Stack view

Example

The following procedure shows how you might use the Call Stack view:

1. Connect to your target.
2. Load the required image. For example, `dhystone.axf`.

-  3. Click the **Locate PC** button on the Debug toolbar to view the source file dhry_1.c.
4. Double-click in the gray margin at line 301 of file dhry_1.c. This sets a simple breakpoint.
5. Click **Run** to start execution.
6. Enter 5000 when prompted for the number of runs. Execution stops when the breakpoint at line 301 is reached. The red box marks the location of the PC when execution stops.
7. Select **Call Stack** from the **View** menu to display the Call Stack view. Figure 13-48 shows an example:

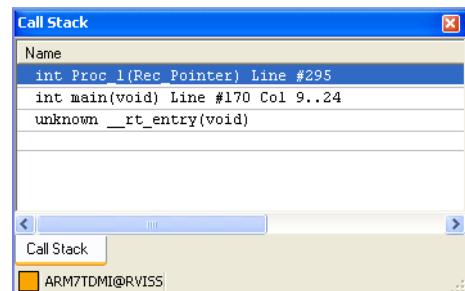


Figure 13-48 Multistatement details in the Call Stack view

8. Continue to step through your program.
9. Monitor changes in the Call Stack view as you step through your program.
10. Double-click on the red marker disc at line 301 of file dhry_1.c. This clears the breakpoint.

See also

- *Loading an executable image* on page 4-4
- *Setting a simple breakpoint* on page 11-13
- *Viewing variables for the current context* on page 13-14
- *Connecting to a target* on page 3-27
- *Operations you can perform on the Call Stack* on page 13-66.

13.15.2 Copying a Call Stack entry

To copy an entry on the Call Stack:

1. Right-click on the required Call Stack entry to display the context menu.
2. Select **Copy** from the context menu.

You can paste the Call Stack entry into an application of your choice, such as a text editor.

13.15.3 Toggling automatic updates of the Call Stack view

By default, the Call Stack view updates automatically. To toggle the automatic update of the Call Stack view:

1. Right-click in the Call Stack view to display the context menu.
2. Select **Automatic Update** from the context menu.

Automatic updating refreshes the Call Stack view when program execution stops at a point in your image, such as at a breakpoint.

Freezing the contents of the Call Stack view

To freeze the contents of the Call Stack view, disable automatic updating. You can manually update the view contents if required.

Freezing the contents of the Call Stack view enables you to compare the contents with those in a second Call Stack view.

See also

- *Updating windows and views when a breakpoint activates* on page 12-4
- *Manually updating the Call Stack view*.

13.15.4 Manually updating the Call Stack view

To manually update the Call Stack view:

1. Disable automatic updates.
2. Right-click in the Call Stack view to display the context menu.
3. Select **Update View** from the context menu.

The Call Stack is updated.

Note

You can also update the Call Stack view when a breakpoint activates.

See also

- *Updating windows and views when a breakpoint activates* on page 12-4
- *Toggling automatic updates of the Call Stack view* on page 13-64.

13.15.5 Displaying the properties of a Call Stack entry

To display the properties of a Call Stack entry:

1. Right-click on the required Call Stack entry to display the context menu.
2. Select **Properties** from the context menu to display the Prompt dialog box. This shows the properties for the chosen Call Stack entry. Figure 13-49 shows an example:

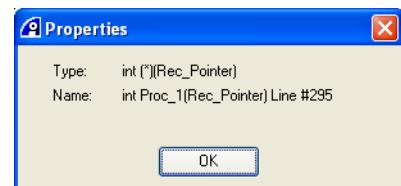


Figure 13-49 Properties of a Call Stack entry

3. Click **OK** to close the dialog box.

13.15.6 Operations you can perform on the Call Stack

You can perform the following operations on the Call Stack:

- move up and down the Call Stack
- change scope to the selected entry
- set a breakpoint at the selected entry
- run to the selected entry.

See also

- *Running to an entry in the Call Stack* on page 8-11
- *Moving up and down the Call Stack* on page 10-14
- *Changing scope to a specific entry in the Call Stack* on page 10-12
- *Setting a breakpoint at the location of a Call Stack entry* on page 11-43.

13.16 Setting watches

There are times when you want to monitor the values of specific variables or expressions in your source code during image execution. You do this by creating watches.

See also:

- *Setting a watch variable*
- *Setting a watch expression* on page 13-68
- *Setting a watch by copying and pasting* on page 13-68
- *Setting a watch by dragging and dropping* on page 13-68
- *Setting a watch from the Symbols view* on page 13-69
- *Editing the name of an existing watch* on page 13-69
- *Deleting a watch* on page 13-69.

13.16.1 Setting a watch variable

To set a watch variable:

1. Connect to your target.

2. Load the required image.

For example, `dhystone.axf`.

3. Either:



- click the **Locate PC** button on the Debug toolbar to view the source file that has the current scope
- open the required source file.

For example, `dhry_1.c`.

4. Right-click on a variable name to display the context menu.

For example, `Run_Index` at line 146 of `dhry_1.c`.

5. Select **Watch** from the context menu.

The variable is added to the list of watches.

See also

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4.

13.16.2 Setting a watch expression

You might want to monitor the result of an expression instead of individual variables.

To set a watch expression:

1. Select **Watches** from the **View** menu to display the Watch view.
 2. Right-click in the Watch view to display the context menu.
 3. Select **Add Watch...** from the context menu to display the watch expression dialog box.
- Figure 13-50 shows an example:

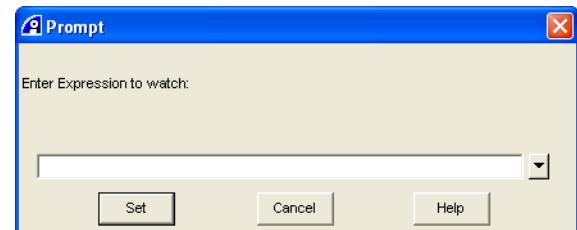


Figure 13-50 Watch expression dialog box

4. Enter the required expression in the dialog box.
5. Click **Set** to set the watch expression.

13.16.3 Setting a watch by copying and pasting

To set a watch by copying and pasting:

1. Select **Watches** from the **View** menu to display the Watch view.
2. Either:
 - copy a variable from the Locals view
 - copy a variable or expression from your source code.
3. Click on the background in the Watch view.
4. Press **Ctrl+V** to paste the variable or expression into the Watch view.

See also

- *Copying and pasting variables from the Locals view to the Watch view* on page 13-18.

13.16.4 Setting a watch by dragging and dropping

To set a watch by dragging and dropping:

1. Select **Watches** from the **View** menu to display the Watch view.
2. Select the required variable or expression in your source code.
3. Click on the selected variable or expression.
4. Drag the variable or expression and drop it into the Watch view.

13.16.5 Setting a watch from the Symbols view

To set a watch from a variable in the Symbols view:

1. Display the Symbols view, and locate the required variable.
 2. Right-click on the variable in the **Variables** tab to display the context menu.
 3. Select **Watch** from the context menu.
- The variable is added to the list of watches.

See also

- *Displaying the list of variables in an image* on page 13-11.

13.16.6 Editing the name of an existing watch

To edit the name of a watch:

1. Double-click in the name of the watch you want to change, or press Enter if the watch is already selected. The name is enclosed in a box with the characters highlighted to show they are selected (pending deletion).
2. Enter the new name, or move the cursor to change the existing expression, or add a cast.
3. Either:
 - press Enter to store the new name
 - press Esc to cancel the change, and revert to the original name.

13.16.7 Deleting a watch

To delete a watch:

1. Select **Watches** from the **View** menu to display the Watch view.
2. Click on the watch to be deleted.
3. Press Delete to delete the watch.

Note

There is no undo for this operation.

13.17 Viewing watches

As you set watches, expressions are added to the watch list. You can view the list of watches using the Watch view.

See also:

- *Examining watches in the Watch view*
- *Toggling automatic updates of the Watch view*
- *Manually updating the Watch view on page 13-71*
- *Performing timed updates for RealMonitor and OS-aware targets on page 13-71*
- *Formatting the display of values for individual watches on page 13-72*
- *Formatting the display of values for all watches on page 13-72*
- *Displaying the properties of a watch on page 13-73*.

13.17.1 Examining watches in the Watch view

To examine the watches you set, select **Watches** from the **View** menu to display the Watch view. Figure 13-51 shows an example Watch view, with several expressions already set.

The screenshot shows a Windows-style application window titled "Watch". The main area is a table with two columns: "Name" and "Value". There are seven rows of data:

Name	Value
Run_Index	0x00000001 <0x6>
Enum_Loc	Ident_2
Int_2_Loc	0x00000009 <0x4>
Int_Glob	0x00000005
Arr_1_Glob	[0x0000E6F0]
Int_1_Loc	0x00000001
Int_3_Loc	0x00000007

Below the table, there are four tabs labeled "Watch1", "Watch2", "Watch3", and "Watch4", with "Watch1" being the active tab. At the bottom of the window, there is a status bar with the text "ARM7TDMI@RVISS".

Figure 13-51 Watch view with watches

The entries correspond to the watches you set, in the order that you set them. Each expression is shown giving the Name and Value. You can expand the column headings by dragging the boundary marker to make the details easier to read.

Expressions are listed in the order they are created. You can drag the column headings to display the full name or value if required. Where an entry contains subentries, for example an array, a plus sign is appended to the name. Click on this to expand the display.

One watched value is an array, shown by the plus sign appended to the variable name. Click the plus sign to expand the view and display the array elements.

Note

If the chosen array is very large, RealView Debugger warns you before expanding the view. Click either **Yes** to expand the array, or **No** to cancel the operation.

13.17.2 Toggling automatic updates of the Watch view

By default, the Watch view updates automatically. To toggle the automatic update of the Watch view:

1. Right-click in the Watch view to display the context menu.
 2. Select **Automatic Update** from the context menu.
- A tick is displayed for the option when it is enabled.

Automatic updating refreshes the Watch view when program execution stops at a point in your image, such as at a breakpoint.

Freezing the contents of the Watch view

To freeze the contents of the Watch view, disable automatic updating. You can manually update the view contents if required.

Freezing the contents of the Watch view enables you to compare the contents with those in a second Watch view.

See also

- *Updating windows and views when a breakpoint activates* on page 12-4
- *Manually updating the Watch view*.

13.17.3 Manually updating the Watch view

To manually update the Watch view:

1. Disable automatic updates.
 2. Right-click in the Watch view to display the context menu.
 3. Select **Update Window** from the context menu.
- The Watch view is updated.

Note

You can also update the Watch view when a breakpoint activates.

See also

- *Updating windows and views when a breakpoint activates* on page 12-4
- *Toggling automatic updates of the Watch view* on page 13-70.

13.17.4 Performing timed updates for RealMonitor and OS-aware targets

If you are using RealMonitor or an OS extension, the Watch can be updated at a specified time interval during program execution.

To perform a timed update of the Watch view:

1. Connect to a target that is configured to run RealMonitor or that is running an OS-aware application.
2. Right-click in the Watch view to display the context menu.
3. Select **Timed Update Period...** from the context menu to display the Timed Update Period dialog box. Figure 13-52 shows an example:

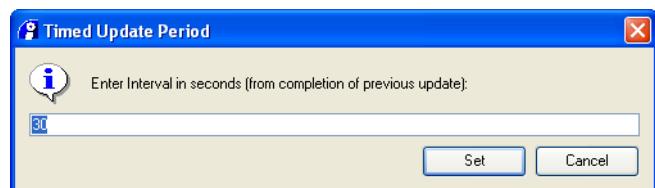


Figure 13-52 Timed Update Period dialog box

4. Enter the interval, in seconds, between window updates. The default is 30 seconds. This value is used only when **Timed Update** is enabled.
5. Right-click in the Watch view to display the context menu.
6. Select **Timed Update** from the context menu.
This option is enabled when:
 - RealMonitor is activated
 - the target is in RSD mode and where supported by the underlying debug target.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring RealMonitor for connections through DSTREAM or RealView ICE* on page 3-43
- *RealView Debugger RTOS Guide*.

13.17.5 Formatting the display of values for individual watches

To format the display of individual watch values:

1. Right-click on the name or value of the variable to display the context menu.
For arrays, structures, or C++ objects:
 - if you right-click on the name, then all values associated with that variable are formatted.
 - if you right-click on the value, then only that value is formatted.
2. Select **Format...** from the context menu to display the selection box. Highlight the required format for the expression from the list of available formats.

————— **Note** —————

If a variable contains multibyte characters, you can choose to view it using ASCII, UTF-8, or Locale encoding.

3. Click **OK**.

13.17.6 Formatting the display of values for all watches

To format the display of values for all watches:

1. Right-click in the Watch view to display the context menu.
2. Select the formatting option you require from the context menu:

Show strings

Displays values of type `char*` and `char[]` (or casted) as strings.

This is enabled by default.

Show integers in hex

Displays all integers in hexadecimal format. Disabling this option displays all integers in decimal.

This is enabled by default.

13.17.7 Displaying the properties of a watch

To display the properties of a watch:

1. Right-click on the required watch to display the context menu.
2. Select **Properties** from the context menu to display the Prompt dialog box. This shows the properties for the chosen watch. Figure 13-53 shows an example:



Figure 13-53 Watch properties

If no properties can be found for the watch, a warning dialog box is displayed.

3. Click **OK** to close the dialog box.

13.18 Viewing statistics for RVISS targets

The statistics for your RVISS target are available by:

- displaying the **CycleCount** tab on the Registers view
- using the STATS CLI command.

See also:

- *Statistics for uncached von Neumann processors*
- *Statistics for uncached Harvard processors* on page 13-75
- *RVISS map file related statistics* on page 13-75
- *Accessing additional processor-specific statistics* on page 13-75
- *Viewing the RVISS cycle count statistics* on page 13-77
- *Accessing the individual cycle count statistics values* on page 13-78
- *Viewing standard cycle count statistics with the STATS command* on page 13-79
- *Example of how to accumulate RVISS statistics* on page 13-80.

13.18.1 Statistics for uncached von Neumann processors

When simulating an uncached von Neumann architecture processor such as the ARM7TDMI, the following information is displayed:

Reference Points

The name you specify to identify each line of statistics that you add.

Instructions The number of program instructions executed.

Core_Cycles Internal processor cycles indicating the time an instruction spends in the execute stage of the pipeline.

S_Cycles The number of sequential cycles performed. The CPU requests transfer to or from the same address, or an address that is a word or halfword after the preceding address.

N_Cycles The number of nonsequential cycles performed. The CPU requests transfer to or from an address that is unrelated to the address used in the preceding cycle.

I_Cycles The number of internal cycles performed. The CPU does not require a transfer because it is performing an internal function (or running from cache).

C_Cycles The number of coprocessor cycles performed.

Total The sum of the S_Cycles, N_Cycles, I_Cycles, and C_Cycles.

See also

- *RVISS map file related statistics* on page 13-75
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the STATS command.
- *RealView ARMulator ISS User Guide*.

13.18.2 Statistics for uncached Harvard processors

When simulating an uncached Harvard architecture processor such as the ARM926EJ-S, the following information is displayed:

Reference Points

The name you specify to identify each line of statistics that you add.

Instructions The number of program instructions executed.

Core_Cycles The total number of processor clock ticks, including stalls because of interlocks and instructions that take more than one cycle.

ID_Cycles Cycles in which both the instruction bus and the data bus were active.

IBus_Cycles Cycles in which the instruction bus was active and the data bus was idle.

Idle_Cycles Cycles in which both the instruction bus and the data bus were idle.

DBus_Cycles

Cycles in which the data bus was active and the instruction bus was idle.

Total The sum of cycles on the memory bus.

See also

- *RVISS map file related statistics*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the STATS command.
- *RealView ARMulator ISS User Guide*.

13.18.3 RVISS map file related statistics

If you use an RVISS map file the display shows additional information, including:

Wait_States The number of wait-states added by the Mapfile component.

True_Idle_Cycles

The number of I_Cycles less the number that are part of an I-S pair. It is only displayed if you set SpotIScycles to True in the RVISS configuration file *peripherals.ami*.

See also

- *RealView ARMulator ISS User Guide*.

13.18.4 Accessing additional processor-specific statistics

If you set Counters=True in the *default.ami* file, then additional processor-specific statistics are available, such as cache hits and misses. You can access these statistics as follows:

- Enter the STATS command to see the RVISS statistics values, which are displayed in the **Cmd** tab of the Output view by default. You can also use the JOURNAL command to save the output to a log file, for example:

```
journal on=RVISS_stats.logstats journal off
```

- Access the `@stats_symbolname` symbols using the CEXPRESSION, FPRINTF, PRINTF, and PRINTVALUE CLI commands, or within a user-defined macro. To get a complete list of the statistics symbols, including the additional processor-specific statistics, enter the following CLI command:
`reginfo,access,match:@stats`
- Display the **CycleCount** tab in the Registers view.

See also

- *Log and journal files* on page 1-44
- Chapter 16 *Using Macros for Debugging*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the CEXPRESSION, FPRINTF, JOURNAL, PRINTF, PRINTVALUE, REGINFO, and STATS commands.
- *RealView ARMulator ISS User Guide*.

13.18.5 Viewing the RVISS cycle count statistics

The RVISS cycle count statistics are displayed in the **CycleCount** tab of the Registers view. What is shown in the tab depends on the processor that you have chosen to simulate.

To view the cycle count statistics:

1. Select **Registers** from the **View** menu to display the Registers view.
2. Click the **CycleCount** tab.

An example for an uncached von Neumann processor, such as an ARM7TDMI, is shown in Figure 13-54.

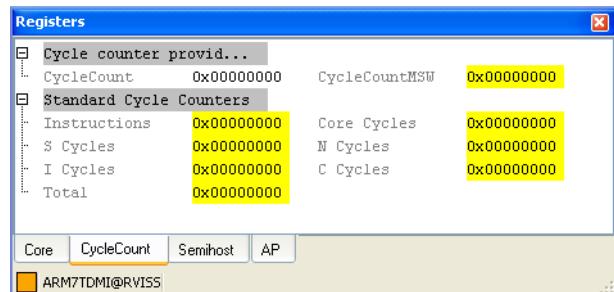


Figure 13-54 CycleCount tab for an uncached von Neumann processor

An example for an uncached Harvard processor, such as an ARM926EJ-S, is shown in Figure 13-55.

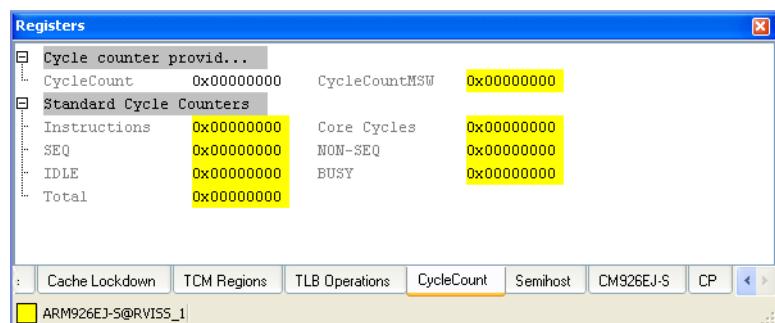


Figure 13-55 CycleCount tab for an uncached Harvard processor

See also

- *RealView ARMulator ISS User Guide*.

13.18.6 Accessing the individual cycle count statistics values

The RVISS cycle count statistics comprise:

- standard cycle counters
- cycle counters provided by the memory callback module.

Standard cycle counters

To access the standard cycle count statistics you can use the symbol `@stats_symbolname`, where `symbolname` is the name of the cycle counter, or the STATS command. A cycle counter name has all the non-alphanumeric characters converted to underscores.

To get a complete list of the standard cycle counter symbols `@stats_symbolname`, enter the following CLI command:

```
> reginfo,access,match:@stats
```

For an uncached von Neumann processor without a map file defined, the following symbols are displayed:

```
Register @stats_Instructions (display name "Instructions")
Register @stats_Core_Cycles (display name "Core_Cycles")
Register @stats__S_Cycles (display name "S_Cycles")
Register @stats__N_Cycles (display name "N_Cycles")
Register @stats__I_Cycles (display name "I_Cycles")
Register @stats__C_Cycles (display name "C_Cycles")
Register @stats_Total (display name "Total")
```

Cycle counters provided by the memory callback module.

For tracing, RealView Debugger uses `@cycle_count`. This is provided by the RVISS memory callback service that watches the bus between the processor and the cache or memory, and provides a reasonable definition of time.

Note

The bus ratio between the inside of the processor and the outside might mean that `@cycle_count` is several times the value of the other clock.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the REGINFO and STATS commands
- *RealView ARMulator ISS User Guide*.

13.18.7 Viewing standard cycle count statistics with the STATS command

Example 13-1 shows statistics for a von Neumann processor, such as an ARM7TDMI:

Example 13-1 RVISS statistics for a von Neumann processor

```
> stats
Ref_Point Instructions Core_Cycles _S_Cycles _N_Cycles _I_Cycles _C_Cycles Total
Ref_Cur    0023adcf    00478cdd   002cb733 0011eb35 0008ea75 00000000 00478d29
```

A group of RealView Debugger internal variables contains statistics relating to your current debugging session.

The first line of statistics shows values accumulated from the beginning of execution of the program you are debugging, and is labeled Ref_Cur.

You can add more lines of statistics, accumulated from later interruptions of program execution. When execution has stopped, to start accumulating a new line of statistics, add a new reference point using the following command:

`STATS.setref reference_point`

— **Note** —

You cannot accumulate statistics in the **CycleCount** tab of the Registers view. The **CycleCount** tab shows the values that correspond to the Ref_Cur reference point.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the STATS command
- *RealView ARMulator ISS User Guide*.

13.18.8 Example of how to accumulate RVISS statistics

For example, you might want to view the counts from the point at which a breakpoint is hit. To do this, you might set a breakpoint that is activated after 10 passes, then create a reference point to show the counts from this point onwards. The following example uses the dhystone image:

1. Connect to an RVISS target on the RealView Instruction Set Simulator Debug Interface.
2. Load the example dhystone image ...\\Debug\\dhystone.axf.
3. Enter the following command to set a breakpoint:
BREAKINSTRUCTION,passcount:10 \DHRY_1\#149:1
4. Start execution.
5. Enter **1000** when prompted for the number of runs.
6. When the breakpoint is activated, create your reference point (iter:10):
STATS,setref iter:10
7. View the current values:

```
> STATS
Ref_Point Instructions Core_Cycles _S_Cycles _N_Cycles _A_Cycles C_Cycles Total
Ref_Cur 00007cb1    0000c703    0000e4b6 00000000 00031279 00000000 0003f72f
iter:10 00000000    00000000    00000000 00000000 00000000 00000000 00000000
```

8. Restart execution.
9. View the current values:

```
> STATS
Ref_Point Instructions Core_Cycles _S_Cycles _N_Cycles _A_Cycles C_Cycles Total
Ref_Cur 00007e97    0000ca13    0000e800 00000000 00031e87 00000000 00040687
iter:10 000001e6    00000310    0000034a 00000000 00000c0e 00000000 00000f58
```

10. You can create additional reference points as required.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the BREAKINSTRUCTION and STATS commands
- *RealView ARMulator ISS User Guide*.

13.19 Viewing the RVISS map related statistics in RealView Debugger

If you have configured RVISS to use a map file, then you can access the map related statistics.

See also:

- *Viewing the map related statistics on the Mapfile tab*
- *Viewing the statistics with CLI commands* on page 13-82.

13.19.1 Viewing the map related statistics on the Mapfile tab

The RVISS map related statistics are displayed in the **Mapfile** tab of the Registers view.

To see the map related statistics:

1. Select **Registers** from the **View** menu to display the Registers view.
2. Click the **Mapfile** tab.

Figure 13-56 shows an example:

Registers			
Base address	Limit address	ReadWriteAbility	Region Name
0x28000000	0x2807FFFF	All	
Width in bytes	ReadWriteAbility	Sequential Reads	0x00000000
0x00000004	All	Sequential Writes	0x00000000
Non-sequential Reads	0x00000000	Time(s)	0x00000000
Non-sequential Writes	0x00000000	Region Name	SSRAM
Time(ns)	0x00000000	Base address	0x24000000
Region Name	SSRAM	Width in bytes	0x00000004
Limit address	0x25FFFFFF	Non-sequential Reads	0x00000000
ReadWriteAbility	All	Non-sequential Writes	0x00000000
Sequential Reads	0x00000000	Time(ns)	0x00000000
Sequential Writes	0x00000000	Region Name	Flash
Time(ns)	0x00000000	Base address	0x2007FFFF
Region Name	Flash	Width in bytes	0x00000004
Base address	0x20000000	ReadWriteAbility	ReadOnly
Width in bytes	0x00000004	Sequential Reads	0x00000000
0x00000004	All	Sequential Writes	0x00000000
Non-sequential Reads	0x00000000	Time(ns)	0x00000000
Non-sequential Writes	0x00000000	Region Name	Boot
Time(ns)	0x00000000	Base address	0x1A000000
Region Name	Boot	Width in bytes	0x00000000

Core CycleCount Mapfile Semihost

ARM7TDMI_MAP@RVISS_3

Figure 13-56 RVISS map related statistics in the Mapfile tab

See also

- *Memory mapping with RVISS models* on page 9-3
- *RVISS map file related statistics* on page 13-75
- *RealView Debugger Target Configuration Guide*
- *RealView ARMulator ISS User Guide*.

13.19.2 Viewing the statistics with CLI commands

From RealView Debugger you can access the map related statistics with the @mapfile_symbolname symbols.

To see a list of these symbols for your model, enter the following CLI command:

```
reginfo,access,match:@mapfile
```

For example, enter the following command to list the variables for the memory region called SRAM:

```
> reginfo,access,match:SRAM
Register @mapfile_SRAM_Address (display name "Base address")
Register @mapfile_SRAM_Limit (display name "Limit address")
Register @mapfile_SRAM_Width (display name "Width in bytes")
Register @mapfile_SRAM_Access (display name "ReadWriteAbility")
Register @mapfile_SRAM_Reads_N (display name "Non-sequential Reads")
Register @mapfile_SRAM_Reads_S (display name "Sequential Reads")
Register @mapfile_SRAM_Writes_N (display name "Non-sequential Writes")
Register @mapfile_SRAM_Writes_S (display name "Sequential Writes")
Register @mapfile_SRAM_Time_ns (display name "Time(ns)")
Register @mapfile_SRAM_Time_s (display name "Time(s)")
Register @mapfile_SRAM_Region_Name (display name "Region Name")
```

See also

- *Memory mapping with RVISS models* on page 9-3
- *RVISS map file related statistics* on page 13-75
- *RealView Debugger Target Configuration Guide*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the REGINFO command
- *RealView ARMulator ISS User Guide*.

13.20 Saving memory contents to a file

You can save all or part of the target memory to a file.

To save the contents of a memory range into a file:

1. Select **Debug** → **Memory/Register Operations** → **Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box. Figure 13-57 shows an example:

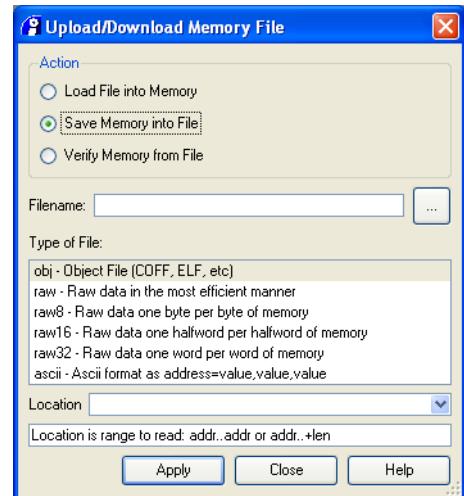


Figure 13-57 Upload/Download file from/to Memory dialog box

2. Specify the operation and set up the controls, as follows:
 - a. Select the **Save Memory into File** radio button. This instructs RealView Debugger to access the specified memory block, read the contents, and write them to the given file.
 - b. In the **File** text box, enter the full path name of the file to use to read/write memory values.
 - c. In the **Type of File** section of the dialog box, select the data type to be used in the specified file where:
 - **obj** specifies an object file in the standard executable target format, for example ARM-ELF for ARM architecture-based targets.
 - **raw** specifies raw data using the most efficient access size for the target.
 - **raw32** specifies a data file as a stream of 32-bit values.
 - **raw16** specifies a data file as a stream of 16-bit values.
 - **raw8** specifies a data file as a stream of 8-bit values.
 - **ascii** specifies a space-separated file of hexadecimal values. A header line is included at the start of the file that describes the file format:
[start, end, size]
where:
— *start* and *end* specifies the address range that is written
— *size* is a character that indicates the size of each value, where b is 8 bits (the default), h is 16 bits and l is 32 bits.
 - d. Specify the region of memory to save, either as:
 - an address range (0x88A0..0x8980)
 - a start address and length (0x88A0..+1000 or main..+1000).

If required, use the drop-down arrow to select a previously used location from the stored list.

3. Click **Apply** to create and write the specified file. If the specified file already exists, you are prompted to overwrite it:
 - Click **Yes** to overwrite the file.
 - Click **No** to cancel the save operation.
4. Click **Close** to close the Upload/Download file to/from Memory dialog box.

See also:

- *Considerations when saving memory contents to a file*
- *Loading the contents of a file into memory* on page 14-19
- *Comparing target memory with the contents of a file* on page 13-85.

13.20.1 Considerations when saving memory contents to a file

Be aware of the following:

- If you are writing memory to a file and the specified file already exists, RealView Debugger warns of this and asks for confirmation before overwriting the file contents.
- RealView Debugger warns you if the memory transfer is going to take a long time to complete. When reading or writing memory contents, you must be aware that:
 - There is no limit on the size of file that RealView Debugger can handle.
 - The time taken to complete the operation depends on the access speeds of your debug target interface.

13.21 Comparing target memory with the contents of a file

You can compare an area of target memory with the contents of a file. For example, you might have a reference file that contains known values for an area of memory, and you want to make sure that the target memory is not corrupted.

Before you can compare target memory with a file, you must have downloaded a memory range into a file as described in *Saving memory contents to a file* on page 13-83. The file must contain at least the memory range you want to compare against.

To verify target memory with the contents of a file:

1. Select **Debug** → **Memory/Register Operations** → **Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box. Figure 13-57 on page 13-83 shows an example.
2. Select **Verify Memory and File**.
3. Specify the file to be compared.
4. In the **Type of File** section of the dialog box, select the data type to be used in the specified file where:
 - **obj** specifies an object file in the standard executable target format, for example ARM-ELF for ARM architecture-based targets.
 - **raw** specifies raw data using the most efficient access size for the target.
 - **raw32** specifies a data file as a stream of 32-bit values.
 - **raw16** specifies a data file as a stream of 16-bit values.
 - **raw8** specifies a data file as a stream of 8-bit values.
 - **ascii** specifies a space-separated file of hexadecimal values. A header line at the start of the file describes the file format:
`[start, end, size]`
 where:
 - *start* and *end* specifies the address range
 - *size* is a character that indicates the size of each value, where **b** is 8 bits, **h** is 16 bits and **l** is 32 bits.
5. Do one of the following:
 - If you selected either the **obj** or **ascii** file type, you can optionally enter a signed offset (in bytes) value. This enables you to skip the specified number of bytes from the start of the file.
 - If you selected any other type of file, specify the start address or range of addresses to be compared.

The resulting addresses must be within the range specified in the memory file.

6. Click **Apply** to compare the file contents with the specified memory block.

7. Click the **Cmd** tab in the Output view to see the results.

If there is a mismatch, the first mismatch is identified and the location reported. Any mismatches after this location are not reported, for example:

```
verifyfile,raw8/gui "C:\Test_files\memory_file_3.mem"=0x8000..0x9000
Mismatch at Address 0x000088B6: 0x8E vs 0x8F
```

8. Click **Close** to close the Upload/Download file from/to Memory dialog box.

See also:

- *Loading the contents of a file into memory* on page 14-19
- *Saving memory contents to a file* on page 13-83.

13.22 Displaying information in a user-defined window

Many CLI commands enable you to send output to a user-defined window.

One possible use of this feature, is in a macro that is attached to a breakpoint. This enables you to view the values of variables and registers when the breakpoint is activated. In this case, you might want to have the breakpoint stop execution so that you can examine the values.

Note

This feature is available only in the RealView Debugger GUI.

See also:

- *Opening a user-defined window*
- *Closing a user-defined window* on page 13-88
- *Clearing a user-defined window* on page 13-88

13.22.1 Opening a user-defined window

To open a user-defined window:

1. At the RealView Debugger CLI prompt in the **Cmd** tab, enter the command:

`Stop> VOPEN windowid`

where *windowid* is an integer in the range 50 to 1024 that identifies the window.

For example:

`Stop> VOPEN 100`

A User window is displayed, shown in Figure 13-58.

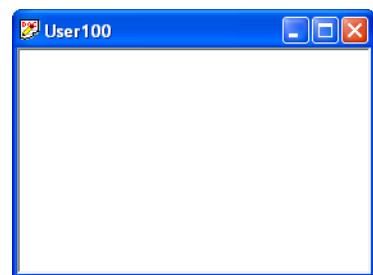


Figure 13-58 User-defined window

2. Specify the *windowid* in any CLI commands that support it.

You can use the commands in scripts or macros, as required.

CLI commands that support user-defined window IDs

The following CLI commands support user-defined window IDs:

- BREAKACCESS
- BREAKEXECUTION
- BREAKINSTRUCTION
- BREAKREAD
- BREAKWRITE
- DCOMMANDS
- DLOADERR
- DOS_resource_list commands (for OS-aware targets)

- DPIPEVIEW
- DTBOARD
- DTBREAK
- DTFILE
- DTRACE
- EXPAND
- FPRINTF
- REGINFO
- SHOW
- VCLEAR
- VCLOSE
- VMACRO
- VOPEN
- VSETC.

Predefined macros that support user-defined window IDs

The following predefined macro supports user-defined window IDs:

- `fwrite.`

See also

- *Setting the execution behavior for a breakpoint* on page 12-9
- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- *CLI commands that support user-defined file IDs* on page 13-90
- *Predefined macros that support user-defined file IDs* on page 13-91
- *Using CLI commands in macros* on page 16-6
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12
 - *Alphabetical predefined macro reference* on page 3-6.

13.22.2 Closing a user-defined window

To close a user-defined window, enter the command:

Stop> `VCLOSE windowid`

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the VCLOSE command.

13.22.3 Clearing a user-defined window

To clear the contents of a user-defined window, enter the command:

Stop> `VCLEAR windowid`

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the VCLEAR command.

13.23 Saving information to a user-defined file

Many CLI commands enable you to send output to a user-defined file.

One possible use of this feature, is in a macro that is attached to a breakpoint. This enables you to view the values of variables and registers when the breakpoint is activated. In this case, you might want to have the breakpoint continue execution.

See also:

- *Opening a user-defined file*
- *Closing a user-defined file* on page 13-92

13.23.1 Opening a user-defined file

To save information to a user-defined file:

1. At the RealView Debugger CLI prompt, enter the command:

`Stop> FOPEN fileid, filename`

where `fileid` is an integer in the range 50 to 1024 that identifies the file.

For example:

`Stop> FOPEN 100, 'C:\myfiles\myfile.txt'`

Alternatively, use the `fopen` predefined macro in a macro, for example:

```
define void copyFile()
{
    int retval;
    // Open data file to read
    retval = fopen(100,"c:\\myfiles\\data_in.txt","r");
    ...
    fclose(100);
}
.
```

2. Specify the `fileid` in any CLI commands or predefined macros that support it.

You can use the commands in scripts or macros, as required.

CLI commands that support user-defined file IDs

The following CLI commands support user-defined file IDs:

- BREAKACCESS
- BREAKEXECUTION
- BREAKINSTRUCTION
- BREAKREAD
- BREAKWRITE
- DCOMMANDS
- DLOADERR
- DOS_resource_list commands (for OS-aware targets)
- DPIPEVIEW
- DTBOARD
- DTBREAK
- DTFILE
- DTRACE
- EXPAND
- FOPEN

- FPRINTF
- REGINFO
- SHOW
- VCLOSE
- VMACRO.

Predefined macros that support user-defined file IDs

The following predefined macros support user-defined file IDs:

- fclose
- fgetc
- fopen
- fputc
- fread
- fwrite.

See also

- *Setting the execution behavior for a breakpoint* on page 12-9
- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- *CLI commands that support user-defined window IDs* on page 13-87
- *Predefined macros that support user-defined window IDs* on page 13-88
- *Using CLI commands in macros* on page 16-6
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12
 - *Alphabetical predefined macro reference* on page 3-6.

13.23.2 Closing a user-defined file

To close a user-defined file, enter the command:

Stop> **VCLOSE *fileid***

If you opened the file with the `fopen` predefined macro, you can use the `fclose` predefined macro.

See also

- *Using CLI commands in macros* on page 16-6
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the VCLOSE command.

13.24 Displaying a list of open user-defined windows and files

You obtain a list of user-defined windows and files that have been opened, either by you or by command scripts that you run.

To display a list of open user-defined windows and files, at the RealView Debugger CLI prompt enter the command:

```
Stop> WINDOW
```

For example:

```
Stop> WINDOW
Num      Type     Name
200      Files    myfile.txt
100      User     User100
Available Terminal Window types: File, User
```

See also:

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the WINDOW command.

Chapter 14

Altering the Target Execution Environment

This chapter describes the RealView® Debugger options that enable you to work with registers and memory interactively during your debugging session. It contains the following sections:

- *About altering the target execution environment* on page 14-2
- *Changing the value of a register* on page 14-3
- *Changing memory contents* on page 14-12
- *Changing the data width on memory accesses* on page 14-18
- *Loading the contents of a file into memory* on page 14-19
- *Changing the stack pointer* on page 14-21
- *Changing the value of a watch* on page 14-23
- *Communicating with a target over DCC* on page 14-24.

14.1 About altering the target execution environment

When target execution stops at specific points, you can examine various items before continuing execution. However, while execution is stopped, you can alter the values of specific items to influence how the target executes when you next start it.

See also:

- *Basic steps required for altering the target execution environment.*

14.1.1 Basic steps required for altering the target execution environment

To alter the target execution environment you must perform the following steps:

1. Specify the point where target execution is to stop, and run the target to that point. You can specify where execution stops by:
 - setting a breakpoint that stops execution
 - running to a specific point
 - stepping.
2. Examine and change the values of specific items. You can change the values of:
 - registers
 - memory locations
 - watches
 - the stack pointer.

You can also read the contents of a file into memory, if required.

3. Start execution again.

See also

- *Running an image to a specific point* on page 8-7
- *Stepping by lines of source code* on page 8-12
- *Stepping by instructions* on page 8-16
- *Stepping until a user-specified condition is met* on page 8-20.
- *Changing the value of a register* on page 14-3
- *Changing memory contents* on page 14-12
- *Loading the contents of a file into memory* on page 14-19
- *Changing the stack pointer* on page 14-21
- *Changing the value of a watch* on page 14-23
- *Chapter 11 Setting Breakpoints*
- *Chapter 13 Examining the Target Execution Environment.*

14.2 Changing the value of a register

Sometimes, you might want to change the value of certain registers when execution stops at a specified point.

Note

You can set breakpoints on memory mapped registers but not on core registers, because breakpoints are set on addresses and core registers do not have addresses.

To change the value of a register, you must stop execution at a specific point of interest.

See also:

- [Changing the value of a register](#)
- [Setting registers with the Interactive Register Setting dialog box on page 14-4](#)
- [Copying and pasting a selected register value on page 14-5](#)
- [Changing values for the CPSR and SPSR registers on page 14-6](#)
- [Changing values for the CP15 Control register on page 14-9](#)
- [Changing debugger internals on page 14-10.](#)

14.2.1 Changing the value of a register

To change the value of a register:

1. Select **Registers** from the **View** menu to display the Registers view. Figure 14-1 shows an example:

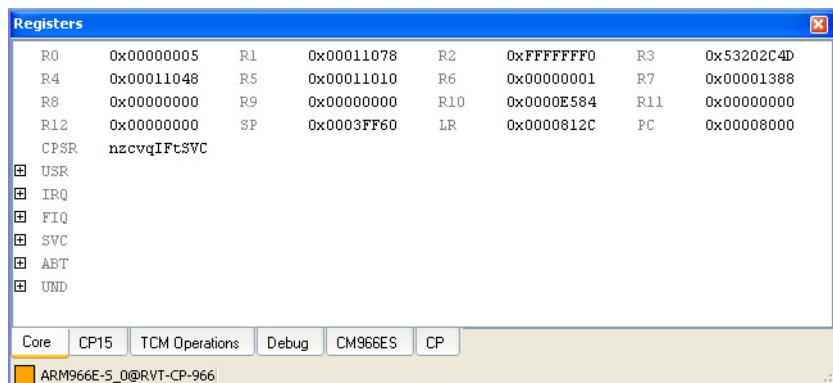


Figure 14-1 Registers view

2. Locate the register that you want to change.
3. Double-click on the register.

The field containing the value changes to allow in-place editing, shown in Figure 14-2.

R2	0x00000000
R6	0x00000000
R10	0x0000E584
LR	0x0000B25C

Figure 14-2 In-place editing of a register

4. Change the value of the register to the required value.
 5. Press Enter to confirm the change. The color of the register value changes to blue.
- Alternatively, press Esc to cancel the change. The register value reverts back to the value before you made any changes.

Changing the value of some registers affects other parts of RealView Debugger:

- If you change the value of the SP register, then the Stack view is updated to start from the new address.
- If you change the value of the PC register, then:
 - The code views change to show the code at the new PC location.
 - The Call Stack view changes to show the new context. Although the **Locals**, **Statics**, and **This** tabs show the variables that are in scope at the new context, the **Call Stack** tab does not have any traceback information.

See also

- *Resetting the PC to the image entry point* on page 10-7
- *Setting the PC to a function* on page 10-9
- *Changing the stack pointer* on page 14-21.

14.2.2 Setting registers with the Interactive Register Setting dialog box

To set register contents with the Interactive Register Setting dialog box:

1. Select **Registers** from the **View** menu to display the Registers view. Figure 14-1 on page 14-3 shows an example.
2. Select **Debug → Memory/Register Operations → Set Register...** from the Code window main menu to display the Interactive Register Setting dialog box. Figure 14-3 shows an example:



Figure 14-3 Interactive Register Setting dialog box

3. Enter the register to change in the Register field. If required, use the drop-down arrow to select a previously used register from the stored list.
For example, enter **@R4**.
4. Enter the value for the register in the Enter New Value field. If required, use the drop-down arrow to select a previously used value from the stored list.
For example, enter **0x4000**.
5. Click **Set** to change the value of the specified register. If required, use the drop-down arrow to select a previously used value from the stored list.

The Log at the bottom of the dialog box is updated to show your change, shown in Figure 14-4 on page 14-5.

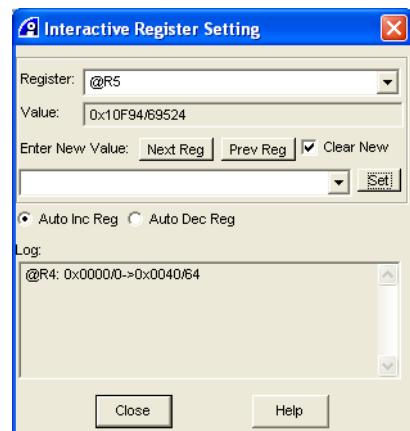


Figure 14-4 Register set in Interactive Register Setting dialog box

- Click **Close** to close the Interactive Register Setting dialog box.

Changing the value of some registers affects other parts of RealView Debugger:

- If you change the value of the SP register, then the Stack view is updated to start from the new address.
- If you change the value of the PC register, then:
 - The code views change to show the code at the new PC location.
 - The Call Stack view changes to show the new context, but does not have any traceback information.
 - The **Locals**, **Statics**, and **This** tabs in the Locals view change as appropriate to show the variables that are in scope at the new context.

See also

- Resetting the PC to the image entry point* on page 10-7
- Setting the PC to a function* on page 10-9
- Changing the stack pointer* on page 14-21.

14.2.3 Copying and pasting a selected register value

To copy and paste a selected register value:

- Right-click on the register containing the value that you want to copy to display the context menu.
- Select **Copy Value** from the context menu.
- Select one or more registers that you want to change.
- Right-click on any selected register to display the context menu.
- Select **Paste Value** from the context menu to paste the value into the selected register. If you selected multiple registers, then the copied value is pasted into all the selected registers.

14.2.4 Changing values for the CPSR and SPSR registers

The CPSR and SPSR register entries include a settings string showing the current settings of the flags in the register, for example:

CPSR nzcvqIFtSVC

You change the flags by using the PSR dialog box.

To change values for the CPSR or an SPSR register:

1. To change a specific SPSR register, expand the appropriate register bank. Otherwise, skip this step. The rest of this procedure describes how to change the CPSR register.
2. Double-click on the CPSR register, shown in Figure 14-5.

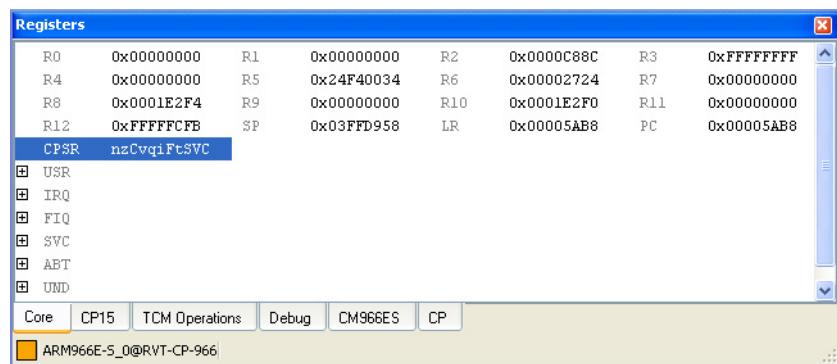


Figure 14-5 CPSR register view

The PSR dialog box is displayed, shown in Figure 14-6.

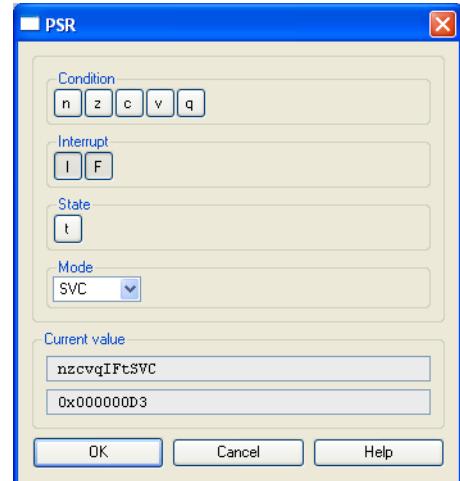


Figure 14-6 PSR dialog box

3. Change the settings as required:
 - To change the Flags, Interrupt or State, click the appropriate button. When a button is depressed, the corresponding bit in the CPSR register is set. The letter in the register field that corresponds to the bit changes to uppercase. That is, uppercase letters identify bits that are set.
 - To change the Mode, select the required mode from the drop-down list.

The new settings are displayed as text and hexadecimal format in the Current value group.
4. Click **OK** to apply the changes. The register value is colored blue.

Processors that support saturating arithmetic instructions

For ARM® architecture-based processors that have the extra Q bit (signifying saturation) in the program status register, the dialog box has a **Q** button in the Flags group. Figure 14-8 on page 14-8 shows an example.

Processors that support Jazelle or Thumb-2EE

For ARM architecture-based processors that support Jazelle® bytecode or Thumb®-2EE there are extra bits:

- For processors that support Jazelle bytecode, a J bit in the program status register signifies Jazelle state. Also the PSR dialog box has a **J** button in the State group.
- For processors that support Thumb-2EE, a T bit in the program status register signifies ThumbEE state. Also the PSR dialog box has a **T** button in the State group.

To set the State for processors that support both Jazelle bytecode and Thumb-2EE, set the button states shown in Table 14-1.

Table 14-1 CPSR and SPSR register T and J bit states

State	T bit state	J bit state
ARM	0 (t)	0 (j)
Thumb	1 (T)	0 (j)
Bytecode	0 (t)	1 (J)
ThumbEE	1 (T)	1 (J)

For Thumb-2EE processors there is also:

- a Control group containing:
 - an **E** button (signifying data endianness)
 - an **A** button (signifying imprecise data abort disable).
- a Greater than or Equal group of buttons for the GE[3:0] bits used by the SIMD instructions.

Figure 14-7 shows an example of the CPSR register for the ARM1136JF-S™. Figure 14-8 on page 14-8 shows the corresponding PSR dialog box.

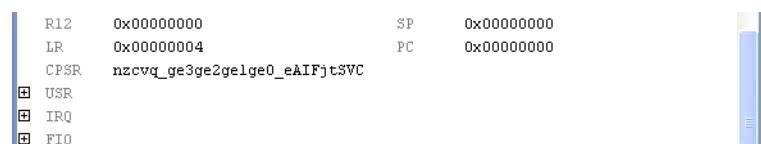


Figure 14-7 CPSR register for the ARM1136JF-S



Figure 14-8 PSR dialog box for the ARM1136JF-S

Processors that support If Then

For ARM architecture-based processors that support If Then, such as Cortex™-A8, there is an extra IT field in the program status register.

Figure 14-9 shows an example of the CPSR register for the Cortex-A8. Figure 14-10 on page 14-9 shows the corresponding PSR dialog box.

RL2	0x00000000	SP	0x00000000
LR	0x55555555	PC	0x00000000
CPSR	nzcvq_DISABLED_ge3ge2ge1ge0_eAIFjtSVC		
⊕	USR		
⊕	IRQ		
⊕	FIQ		

Figure 14-9 CPSR register for the Cortex-A8

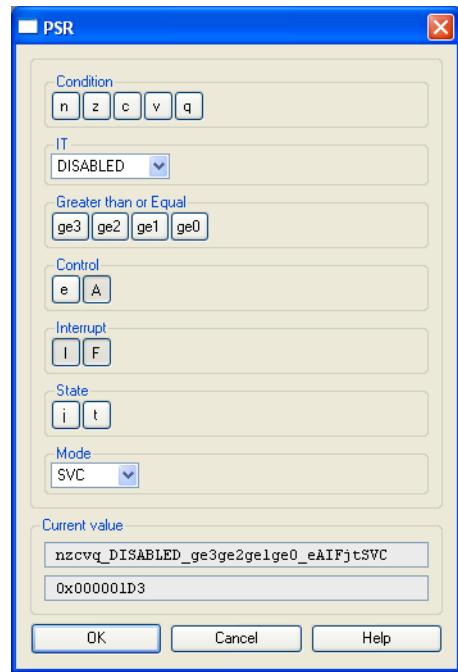


Figure 14-10 PSR dialog box for the Cortex-A8

14.2.5 Changing values for the CP15 Control register

The method for changing the value of the CP15 Control register depends on the architecture of the target.

ARMv5 and later architectures

For ARMv5 and later architectures, the CP15 Control register is displayed in the Registers view as a string that represents the bit pattern. Figure 14-11 shows the CP15 registers for an ARM1136JF-S processor.

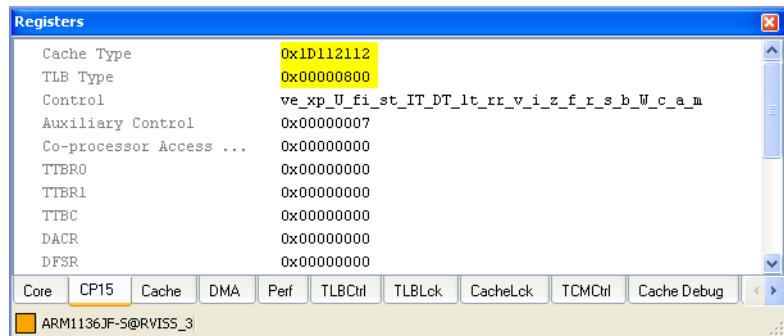


Figure 14-11 ARM1136JF-S CP15 Control register in Registers view

To change the CP15 register:

1. Double-click on the CP15 Control register.

The CP15 Control dialog box is displayed, shown in Figure 14-12 on page 14-10.

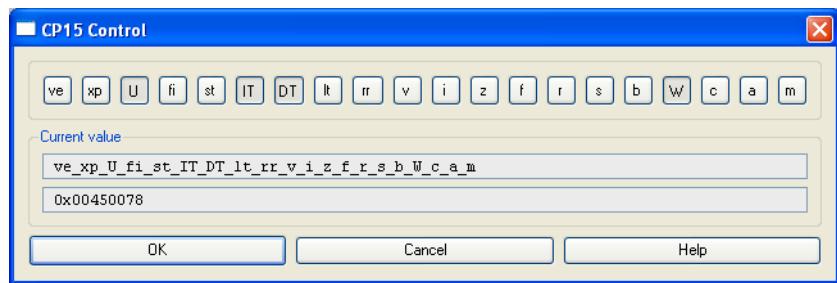


Figure 14-12 CP15 Control dialog box

2. Click the appropriate button to change the settings as required.

When a button is depressed, the corresponding bit in the CP15 Control register is set. The letter in the register field that corresponds to the bit changes to uppercase. That is, uppercase letters identify bits that are set.

The new settings are displayed as text and hexadecimal format in the Current value group.

3. Click **OK** to apply the changes. The register value is colored blue.

ARMv4 and earlier architectures

For ARMv4 and earlier architectures, the CP15 Control register is displayed as a bitmap value (in hexadecimal by default), which you can edit directly. See the Technical Reference Manual for your target processor, for details of the CP15 Control register bit map.

14.2.6 Changing debugger internals

Debugger internals appear in tabs in the Registers view. The tabs that appear depend on your debug target. You can change the values of these internals if required. However, be aware that changing some debugger internals is not recommended, or has other side-effects.

Changing SVC numbers

It is strongly recommended that you do not change the semihosting ARM and Thumb *SuperVisor Call* (SVC) numbers. If you do, you must:

1. Change all the code in your application, including library code, to use the new SVC number.
2. Change the ARM SVC and Thumb SVC settings to use the new SVC numbers.

To do this for a connection, change the `Arm_svc_num` and `Thumb_svc_num` settings in the Connection Properties.

Enabling or disabling semihosting (hardware targets)

For a hardware target connection, you can temporarily override the semihosting setting in the Connection Properties of the related Debug Configuration.

Note

For non ARMv7-M processors with the semihosting vector set to the default (0x8), you cannot enable semihosting if the SVC vector catch is enabled.

To do this:

1. Select **Registers** from the **View** menu to display the Registers view.

2. Click the **Debug** tab.
3. Select either **True** or **False** from the Semihosting enabled drop-down list.

Note

In a multiprocessor configuration, this affects the current connection only.

You can also set the variable `semihost_enabled` directly using the CEXPRESSION CLI command, for example:

```
cexpression @semihost_enabled=1
```

Set the variable to **1** to temporarily enable semihosting, or **0** to temporarily disable semihosting.

To permanently set the semihosting state for a connection, change the Enabled setting in the Semihosting group of the Connection Properties for the required Debug Configuration.

Enabling or disabling semihosting (RVISS)

For a target connection through RVISS, you can temporarily override the semihosting setting in the Connection Properties of the related Debug Configuration. To do this:

1. Select **Registers** from the **View** menu to display the Registers view.
2. Click the **Semihost** tab.
3. Set the Semihosting State value to either **1** (True) or **0** (False).

You can also set the variable `semihost_state` directly using the CEXPRESSION CLI command, for example:

```
cexpression @semihost_state=1
```

See also

- *Viewing debugger internals* on page 13-33
- *Standard semihosting behavior* on page 13-34
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring Semihosting* on page 3-29
 - *Appendix A Connection Properties Reference*.
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the CEXPRESSION command.

14.3 Changing memory contents

You might want to change the contents of various memory locations to see how image execution is affected.

To change the value of a memory location, you must stop execution at a specific point of interest. You can specify where execution stops by:

- setting a breakpoint that stops execution
- running to a specific point
- stepping.

See also:

- *Changing the value in a memory location*
- *Setting memory with the Interactive Memory Settings dialog box* on page 14-15
- *Filling memory with a pattern* on page 14-16
- *Running an image to a specific point* on page 8-7
- *Stepping by lines of source code* on page 8-12
- *Stepping by instructions* on page 8-16
- *Stepping until a user-specified condition is met* on page 8-20
- *Changing the value of a register* on page 14-3
- *Setting registers with the Interactive Register Setting dialog box* on page 14-4
- *Copying and pasting a selected register value* on page 14-5
- *Changing values for the CPSR and SPSR registers* on page 14-6
- *Changing values for the CP15 Control register* on page 14-9
- *Changing debugger internals* on page 14-10
- Chapter 11 *Setting Breakpoints*.

14.3.1 Changing the value in a memory location

To change memory contents:

1. Display the Memory view, if it is not visible, and set the start address of the memory area you want to view. Figure 14-13 shows an example.

Figure 14-13 Memory view showing address contents

2. Double-click on the value at the required memory location.
If the ASCII view is displayed, select the character corresponding to the location to enter an ASCII character in that location.
3. Enter the new value for the location. If you press Esc instead of Enter, then any changes you made in the highlighted field are ignored.

Note

You can enter data values in a format that is different from the display format.

The Memory view is updated, shown in Figure 14-14.

Start address	Columns	Data sizes	Format
0x0000886A	Auto column	Default	Hexadecimal
0x0000886A	+0 +1 +2 +3 +4 +5 +6 +7 +8	R Y S T O N E P	
0x00008873	0x52 0x59 0x53 0x54 0x4F 0x4E 0x45 0x20 0x50	R O G R A M , 2	
0x0000887C	0x27 0x4E 0x44 0x20 0x53 0x54 0x52 0x49 0x4E	' N D S T R I N	
0x00008885	0x47 0x00 0x00 0xF0 0xE6 0x00 0x00 0x44 0x48	G D H	
0x0000888E	0x52 0x59 0x53 0x54 0x4F 0x4E 0x45 0x20 0x50	R Y S T O N E P	
0x00008897	0x52 0x4F 0x47 0x52 0x41 0x4D 0x2C 0x20 0x33	R O G R A M , 3	
0x000088A0	0x4E 0x52 0x44 0x20 0x53 0x54 0x52 0x49 0x4E	' R D S T R I N	
0x000088A9	0x47 0x00 0x00 0xCC 0xE6 0x00 0x00 0x45 0x78	G E X	
0x0000886A			

Figure 14-14 Memory view showing changed value

Example of changing memory values with the Memory view

The following procedure shows how you can change memory locations in the Memory view:

1. Load the debug version of the dhystone.axf image.
2. Click the **Locate PC** button on the Debug toolbar to view the source file dhry_1.c.
3. Double-click in the gray margin at line 301 of dhry_1.c. This sets a simple breakpoint.
4. Click **Run** to start execution.
5. Enter **5000** when asked for the number of runs. The program starts and then stops when execution reaches the breakpoint at line 301. The red box marks the location of the PC when execution stops.
6. Use the toolbar in the Memory view, to set the start address to **0x886A**.
7. Right-click on a memory cell and select Show ASCII to display the ASCII values, shown in Figure 14-15.

Start address	Columns	Data sizes	Format
0x0000886A	Auto column	Default	Hexadecimal
0x0000886A	+0 +1 +2 +3 +4 +5 +6 +7 +8	R Y S T O N E P	
0x00008873	0x52 0x59 0x53 0x54 0x4F 0x4E 0x45 0x20 0x50	R O G R A M , 2	
0x0000887C	0x27 0x4E 0x44 0x20 0x53 0x54 0x52 0x49 0x4E	' N D S T R I N	
0x00008885	0x47 0x00 0x00 0xF0 0xE6 0x00 0x00 0x44 0x48	G D H	
0x0000888E	0x52 0x59 0x53 0x54 0x4F 0x4E 0x45 0x20 0x50	R Y S T O N E P	
0x00008897	0x52 0x4F 0x47 0x52 0x41 0x4D 0x2C 0x20 0x33	R O G R A M , 3	
0x000088A0	0x27 0x52 0x44 0x20 0x53 0x54 0x52 0x49 0x4E	' R D S T R I N	
0x000088A9	0x47 0x00 0x00 0xCC 0xE6 0x00 0x00 0x45 0x78	G E X	
0x0000886A			

Figure 14-15 Example memory display

The memory locations **0000889F-000088A2** contain the four hexadecimal values **0x33**, **0x27**, **0x52**, and **0x44** corresponding to the string **3'RD**.

8. Change the contents of location 0000889F as follows:
 - a. Double-click on the value at 0000889F, that is 0x33.
 - b. Enter 'N (or the equivalent hexadecimal value **0x4E**). If you enter the value as a character, you must include the initial quote.

The new value is displayed in blue and the ASCII value changes from 3 to N.
9. Change the value at the location 000088A0, as follows:
 - a. Click on the ASCII character ' corresponding to the location 000088A0.
 - b. Type lowercase **o**. The character is inserted, and cursor moves to the character at the next location:
 - the new value is displayed in blue
 - the value at location 0000889F is colored light blue to show that it was changed on the previous update. Figure 14-16 shows an example:

0x0000888E	0x52 0x59 0x53 0x54 0x4F 0x4E 0x45 0x20 0x50	RYSTONE P
0x00008897	0x52 0x4F 0x47 0x52 0x41 0x4D 0x2C 0x20 0x4E	R O G R A M , N
0x000088A0	0x6F 0x20 0x32 0x20 0x53 0x54 0x52 0x49 0x4E o	D STRIN

Figure 14-16 Memory updated

10. Type a space followed by 2:
 - the new value at location 000088A2 is displayed in blue
 - the value at location 000088A1 is colored light blue to show that it was changed on the previous update
 - the value at location 000088A0 is colored black.

The string 3'RD has been replaced by No 2. Figure 14-17 shows an example:

0x0000888E	0x52 0x59 0x53 0x54 0x4F 0x4E 0x45 0x20 0x50	RYSTONE P
0x00008897	0x52 0x4F 0x47 0x52 0x41 0x4D 0x2C 0x20 0x4E	R O G R A M , N
0x000088A0	0x6F 0x20 0x32 0x20 0x53 0x54 0x52 0x49 0x4E o	2]STRIN

Figure 14-17 Memory updated

11. Double-click on the red marker disc at line 301 of dhry_1.c. This clears the breakpoint.

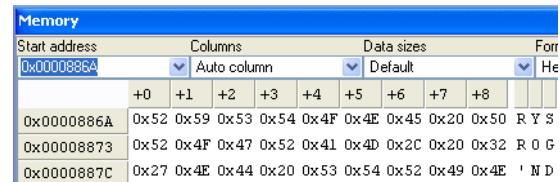
See also

- *Setting a simple breakpoint* on page 11-13
- *Viewing memory contents* on page 13-39
- *MMU page tables views* on page 13-46
- *Managing the display of memory in the Memory view* on page 13-50.

14.3.2 Setting memory with the Interactive Memory Settings dialog box

To set memory contents with the Interactive Memory Setting dialog box:

1. Display the Memory view, if it is not visible.
2. Right-click on a memory address to display the context menu.
3. Select **Set Start Address...** from the context menu to display the Memory view toolbar. The Start Address field is selected. Figure 14-18 shows an example:



The screenshot shows a Windows-style window titled "Memory". At the top, there are dropdown menus for "Start address" (set to 0x0000088A), "Columns" (set to "Auto column"), "Data sizes" (set to "Default"), and "Format" (set to "Hex"). Below this is a table with columns labeled +0, +1, +2, +3, +4, +5, +6, +7, +8. The first row contains the address 0x00000886A followed by its byte values: 0x52, 0x59, 0x53, 0x54, 0x4F, 0x4E, 0x45, 0x20, 0x50. To the right of the table are icons for "R", "Y", "S", "G", "O", and "D". The second row contains the address 0x000008873 followed by its byte values: 0x52, 0x4F, 0x47, 0x52, 0x41, 0x4D, 0x2C, 0x20, 0x32. The third row contains the address 0x00000887C followed by its byte values: 0x27, 0x4E, 0x44, 0x20, 0x53, 0x54, 0x52, 0x49, 0x4E.

Figure 14-18 Start Address field in Memory view

4. Enter **0x000088A0** as the new start address.
5. Select **Debug → Memory/Register Operations → Set Memory...** from the main menu to display the Interactive Memory Setting dialog box. Figure 14-19 shows an example:

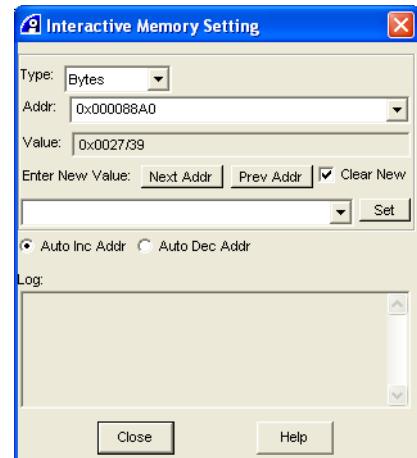


Figure 14-19 Interactive Memory Setting dialog box

6. Enter the memory location to change in the Addr: field, if you want to change the value at a different location. If required, use the drop-down arrow to select a previously used addresses from the stored list.
7. Enter the value for the memory location in the Enter New Value: field. If required, use the drop-down arrow to select a previously used value from the stored list. For example, enter **0x40**.
8. Click **Set** to change the value of the specified memory location. If required, use the drop-down arrow to select a previously used value from the stored list. The Log is updated to show your change, shown in Figure 14-20 on page 14-16.

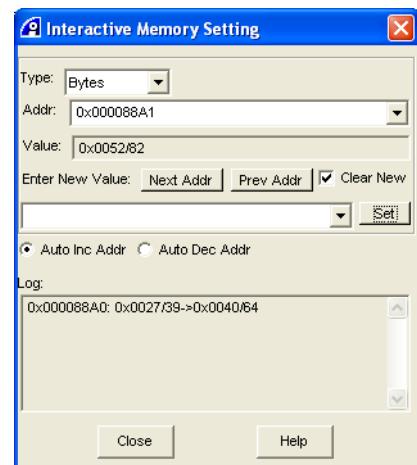


Figure 14-20 Memory set in Interactive Memory Setting dialog box

- Click **Close** to close the Interactive Memory Setting dialog box.

Changed values are displayed in the Memory view in the usual way. That is, updated values are displayed in blue or light blue, depending on when they last changed.

14.3.3 Filling memory with a pattern

To fill a specified area of memory with a predefined pattern:

- Select **Memory** from the **View** menu to display the Memory view.
- Enter **0x88A0** in the Start Address field to view the memory contents at that location.
- Right-click on a memory cell in the Memory view to display the context menu.
- Select **Fill Memory...** from the context menu to display the Fill Memory dialog box.

The fields in the Fill Memory dialog box are populated as follows:

- the Start address field is set to the last address you entered in the Start address field of the Memory view
- the value for the End address and Length depend on the option you have selected for Display sizes in the Memory view.

Figure 14-21 shows an example:

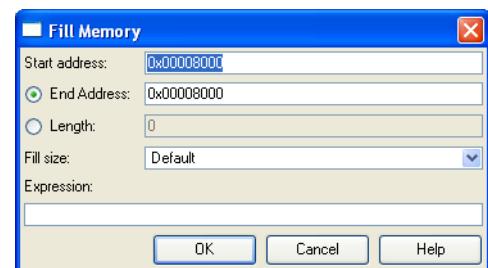


Figure 14-21 Fill Memory dialog box

- Enter the start address of the memory range to be filled in the Start Address: field. For example, enter **0x88A0**.
- Do one of the following:
 - To specify the memory region as a length (number of locations), select the **Length:** and enter the length. For example, enter **14** (decimal).

The specified length must be given relative to the data type given in the Fill Size field.

- To specify the end address of the memory region, select the **End Address:** check box, and enter the address that marks the end of the memory block.
7. Select the access size to be used in the Fill Size: field. The options are:
 - default indicates the native format for the debug target
 - 8 bits indicates support for 8-bit signed and unsigned byte form
 - 16 bits indicates support for 16-bit signed and unsigned halfwords
 - 32 bits indicates support for 32-bit signed and unsigned words.
 8. Enter the pattern to be used as the fill in the Expression: field. For example, enter "**AB**", "**String**",**0**, or **0,1,0,1,0**.

———— Note ————

Strings must be enclosed in single or double quotes.

9. Click **OK** to confirm your settings and close the Fill Memory dialog box. Memory contents are rewritten and the Memory view is automatically updated with changed values displayed in blue.

Considerations when filling memory with a pattern

When filling memory with a pattern, you must be aware of the following:

- No C-style zero terminator byte is written to memory after a specified string. To write a NUL-terminated string, add a zero value expression after the string, for example:
"Test Message",**0**
- You cannot use an empty string to add a NUL character.
- Use the **/8** qualifier if you want to write the characters of a string to consecutive bytes.
- All expressions in an expression string are padded or truncated to the size specified by the Size value if they do not fit the specified size evenly.
- If the number of values in an expression string is less than the number of bytes in the specified address range, RealView Debugger repeats the pattern to fill the remaining number of blocks specified. For example, if you specify a pattern of 10 bytes and a fill area of 16 bytes, RealView Debugger repeats the pattern to fill the remaining six bytes.
- If more values are given than can be contained in the specified address range, excess values are ignored.
- If a pattern is not specified, RealView Debugger displays an error message.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the FILL, SETMEM, and TEST commands.

14.4 Changing the data width on memory accesses

By default, RealView Debugger makes word-size accesses when reading or writing memory. You can change this temporarily in your memory map, or more permanently by configuring your target connection.

See also:

- *Editing a memory map entry* on page 9-20
- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

14.5 Loading the contents of a file into memory

You can load all or part of the contents of a file into target memory.

Before you can load the contents of a file into target memory, you must have downloaded a memory range into a file:

- for binary file types (raw, raw8, and raw16), the file must contain at least the memory range you want to load
- for other file types (obj and ascii), the file is read from the beginning, but it is written to memory starting at the address specified in the file, plus the offset you specify.

To load the contents of a file into memory:

1. Select **Debug → Memory/Register Operations → Upload/Download Memory file...** from the Code window main menu to display the Upload/Download file from/to Memory dialog box. Figure 14-22 shows an example:

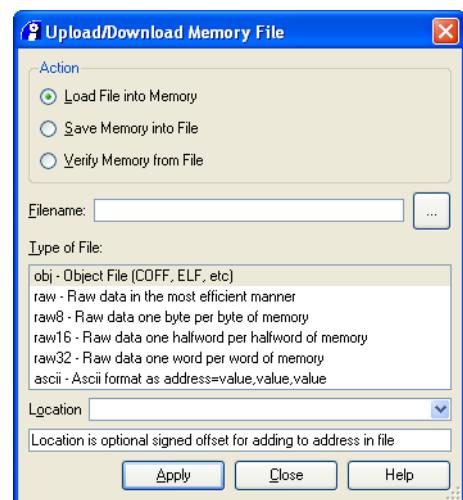


Figure 14-22 Upload/Download file from/to Memory dialog box

2. Specify the operation and set up the controls, as follows:
 - a. Select the **Load File into Memory** radio button.
 - b. In the File: field, enter the full path name of the file to use to read/write memory values.
 - c. In the Type of File section of the dialog, select the data type that you used when the specified file was saved:
 - obj specifies an object file in the standard executable target format, for example ARM-ELF for ARM architecture-based targets.
 - raw specifies raw data using the most efficient access size for the target.
 - raw32 specifies a data file as a stream of 32-bit values.
 - raw16 specifies a data file as a stream of 16-bit values.
 - raw8 specifies a data file as a stream of 8-bit values.
 - ascii specifies a space-separated file of hexadecimal values. A header line at the start of the file describes the file format:
[start,end,size]
where:
— start and end specifies the address range

- *size* is a character that indicates the size of each value, where b is 8 bits, h is 16 bits and l is 32 bits.
- d. In the Location: field, specify the start location of the memory block where the contents are to be written:
- If the file type is obj, then you have only to specify a start location, for example:
 - 0x88A0
 - main
 - If the file type is ascii, then specify a signed offset from the start location that is stored in the ASCII file.
For example, if the stored location is 0x8000 and you want to load the file starting at 0x9000, then enter **0x1000**.
 - If the data type is one of raw, raw8, raw16, or raw32, then you can enter:
 - a start address (0x88A0)
 - an address range (0x88A0..0x8980)
 - a start address and length (0x88A0..+1000 or main..+1000).
- If you specify an address range or a start address and length that is greater than the size of the file, then RealView Debugger stops writing to memory when the end of the file is reached. For example, if 50 bytes of memory are stored in the file, and you specify a length of 100, then only 50 bytes are written to memory.
- If required, use the drop-down arrow to select a previously used location from the stored list.
3. Click **Apply** to load the memory from the specified file.
If you are loading to Flash, then the Flash Memory Control dialog box is displayed.
 4. Click **Close** to close the Upload/Download file to/from Memory dialog box.

See also:

- *Saving memory contents to a file* on page 13-83
- *Comparing target memory with the contents of a file* on page 13-85
- *Writing a binary to Flash* on page 6-4.

14.6 Changing the stack pointer

The address of the stack is determined from SP register. You can change the address of the stack by modifying the SP register.

To change the stack pointer, you must stop execution at a specific point of interest. You can specify where execution stops by:

- setting a breakpoint that stops execution
- running to a specific point
- stepping.

To change the stack pointer:

1. Select **Registers** from the **View** menu to display the Registers view. Figure 14-23 shows an example:

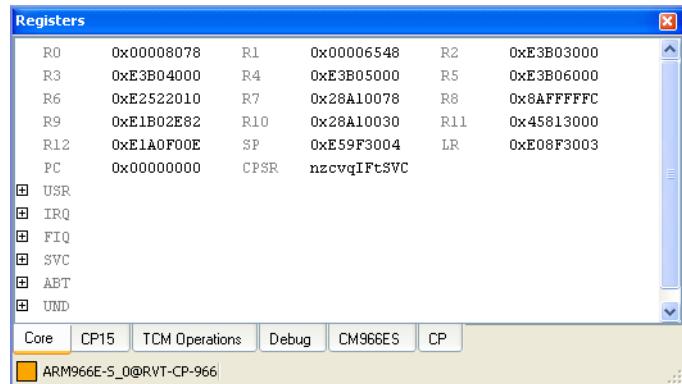


Figure 14-23 Registers view

2. Double-click on the SP register. The field containing the value changes to allow in-place editing, shown in Figure 14-24.

R7	0x28A10078
R10	0x28A10030
SP	0xE59F3004
CPSR	nzcvqIFtSVC

Figure 14-24 In-place editing of the SP register

3. Change the value of the SP register to the required value, for example, 0x01FFD900. Alternatively, press Esc to cancel the change. The register value reverts back to the value before you made any changes.
4. Press Enter to confirm the change. If the Stack view is visible, then the Stack view is updated and starts at the new SP register address, shown in Figure 14-25 on page 14-22.

Stack	
0x01FFD900	SP 0x000000E0
0x01FFD904	0x03E0FF01
0x01FFD908	0x8203E0FF
0x01FFD90C	0xE0FF0300
0x01FFD910	0x03E0FF03
0x01FFD914	0x80030080
0x01FFD918	0xE0FF0300
0x01FFD91C	0x00E0FF01
0x01FFD920	0x80008000
0x01FFD924	0xC0713800
0x01FFD928	0x38C07138
0x01FFD92C	0x7138C071
0x01FFD930	0xF67138C0
0x01FFD934	0x3FF87138
0x01FFD938	0xF13FC0F1
0x01FFD93C	0xC07138C0
0x01FFD940	0x38C07138
0x01FFD944	0x7138C071

Figure 14-25 Stack view for new SP register address

See also:

- *Running an image to a specific point* on page 8-7
- *Stepping by lines of source code* on page 8-12
- *Stepping by instructions* on page 8-16
- *Stepping until a user-specified condition is met* on page 8-20
- Chapter 11 *Setting Breakpoints*.

14.7 Changing the value of a watch

With a watch set, you might want to change the value. You can use various methods to change values of watches in the Watch view.

To change the value of a watch, you must stop execution at a specific point of interest. You can specify where execution stops by:

- setting a breakpoint that stops execution
- running to a specific point
- stepping.

To set a watch value:

1. Click on the value of the watch you want to change.
The value is enclosed in a box with the characters highlighted in green to show they are selected (pending deletion).
2. Enter the new value, or move the cursor to the left to change part of the existing value.
3. Either:
 - press Enter to store the new value
 - press Esc to cancel the change, and revert to the original value.

See also:

- *Running an image to a specific point* on page 8-7
- *Stepping by lines of source code* on page 8-12
- *Stepping by instructions* on page 8-16
- *Stepping until a user-specified condition is met* on page 8-20
- Chapter 11 *Setting Breakpoints*.

14.8 Communicating with a target over DCC

The *Debug Communications Channel* (DCC) enables you to pass information between the target and RealView Debugger over a JTAG connection, without stopping program flow or entering debug state. RealView Debugger provides a Comms Channel view that enables you to interact with a target over the DCC.

See also:

- *Sending information to the target*
- *Receiving information from the target* on page 14-26
- *Saving input to a log file* on page 14-27
- *Changing the format of data displayed in the Data log pane* on page 14-27
- *Showing and hiding data sent to the target* on page 14-27
- *Clearing the Data log pane* on page 14-28.

14.8.1 Sending information to the target

To use the Comms Channel view to send information to the target:

1. To see how the Comms Channel view works:
 - a. Build the `inchan.axf` image. See the `readme.txt` file in `install_directory\RVDS\Examples\...\main\dcc`.
 - b. Connect to your target.
2. Select **Comms Channel** from the **View** menu to display the Comms Channel view. Figure 14-26 shows an example:



Figure 14-26 Comms Channel view

3. Load the required image.

For this example, load the image:

`install_directory\RVDS\Examples\...\main\dcc\inchan.axf`

4. Specify the information to be sent.

To enter information directly in the Comms Channel view:

- a. Type the required information in the Data entry field.
For example, type **And goodbye!**.

- b. Click **Send**.

The data is sent to the target, and is also displayed as grey text in the Data log pane.

To load information from a text file:

- a. Right-click in the Data log pane to display the context menu.
- b. Select **Read Input from File...** from the context menu. The Choose File to Read Input From dialog box is displayed.

- c. Locate the file containing the information to be sent. In this example, locate the file:
install_directory\RVDS\Examples\...\main\dcc\input
- d. Click **Open**.
5. Select **Memory** from the **View** menu to display the Memory view. Figure 14-27 shows an example:

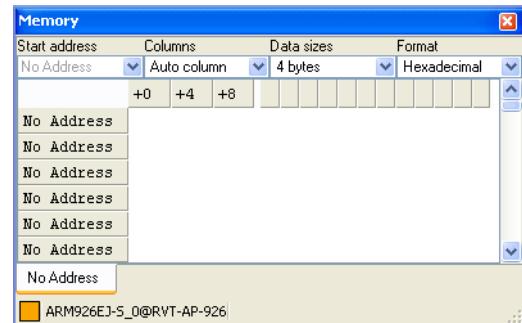
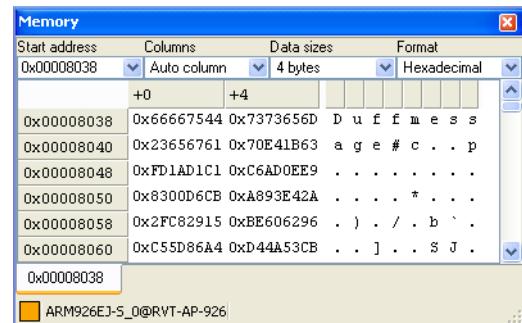


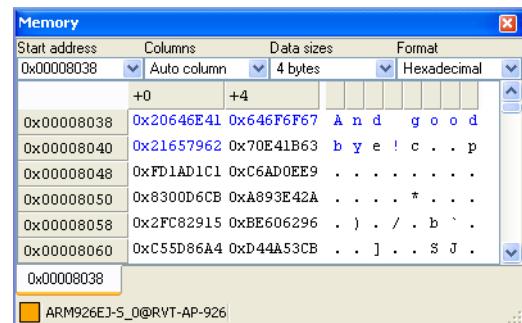
Figure 14-27 Memory view

6. In the **inchan.s** tab of the Source view, double-click on the **indata** label.
7. Copy and paste the **indata** label into the Start Address field of the Memory view, then press Enter. The memory at the address of **indata** shows that defined by the image. Figure 14-28 shows an example:

Figure 14-28 Data defined for **indata** in image

8. Click **Run** to start execution.

The data at the address of **indata** changes to that shown in Figure 14-29.

Figure 14-29 Data sent to **indata** location

See also

- *Comms Channel view* on page 1-13

- *Connecting to a target* on page 3-27
- *Loading an executable image* on page 4-4
- *Starting and stopping image execution* on page 8-4
- the chapter that describes the Debug Communications Channel in *ARM® Compiler toolchain Developing Software for ARM® processors*.

14.8.2 Receiving information from the target

To use the Comms Channel view to receive information from the target:

1. To see how the Comms Channel view works:
 - a. Build the outchan.axf images. See the readme.txt file in *install_directory\RVDS\Examples\...\main\dcc*.
 - b. Connect to your target.
2. Select **Comms Channel** from the **View** menu to display the Comms Channel view. Figure 14-30 shows an example:



Figure 14-30 Comms Channel view

3. Load the required image.

For this example, load the image:

install_directory\RVDS\Examples\...\main\dcc\outchan.axf

4. Click **Run** to start execution.

The data is received from the target, and is displayed as black text in the Data log pane of the Comms Channel view. Figure 14-31 shows an example:

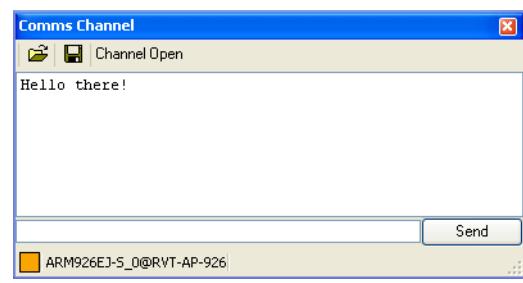


Figure 14-31 Data received in Comms Channel view

See also

- *Comms Channel view* on page 1-13
- *Connecting to a target* on page 3-27

- *Loading an executable image* on page 4-4
- *Starting and stopping image execution* on page 8-4
- the chapter that describes the Debug Communications Channel in *ARM® Compiler toolchain Developing Software for ARM® Processors*.

14.8.3 Saving input to a log file

To save input to a log file:

1. Right-click in the Data log pane to display the context menu.
2. Select **Log Input to File...** from the context menu. The Choose File to Log Input to dialog box is displayed.
3. Locate the directory where you want to store the log file.
4. Enter a filename for the log file.
5. Click **Save**.

14.8.4 Changing the format of data displayed in the Data log pane

To change the format of data displayed in the Data log pane:

1. Right-click in the Data log pane to display the context menu.
2. Select the format type from the **Format** submenu.

See also

- *Comms Channel view* on page 1-13.

14.8.5 Showing and hiding data sent to the target

To show or hide data sent to the target:

1. Right-click in the Data log pane to display the context menu.
2. Select **Local Echo** from the context menu.

When the tick mark is not visible, only data received from the target is now visible.

Note

The default setting is to show data sent to the target.

See also

- *Comms Channel view* on page 1-13.

14.8.6 Clearing the Data log pane

To clear the Data log pane of the Comms Channel view:

1. Right-click in the Data log pane to display the context menu.
2. Select **Clear** from the context menu.

See also

- *Comms Channel view* on page 1-13.

Chapter 15

Debugging with Command Scripts

This chapter describes how to use scripts to run RealView® Debugger CLI commands, to enable you to automate debugging operations. It includes:

- *About debugging with command scripts* on page 15-2
- *Changing output buffering behavior* on page 15-4
- *Creating a log file for use as a command script* on page 15-5
- *Creating log and journal files at start-up* on page 15-6
- *Closing log and journal files* on page 15-7
- *Using macros in command scripts* on page 15-8
- *Running command scripts* on page 15-9
- *Creating a script that writes information to a user-defined window* on page 15-12
- *Creating a script that accesses a user-defined file* on page 15-13.

15.1 About debugging with command scripts

RealView Debugger enables you to use command scripts containing RealView Debugger CLI commands to carry out debugging tasks without user intervention. The commands are actioned as though they are being entered from the keyboard.

See also:

- *Logging and journaling*
- *Command scripts*
- *Macros* on page 15-3.

15.1.1 Logging and journaling

RealView Debugger enables you to save CLI commands that you enter or that are generated by RealView Debugger, and various messages that are displayed during a debugging session. The information that is saved depends on the type of file that is opened.

See also

- *Types of log and journal files* on page 1-44
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 2 *RealView Debugger Commands*.

15.1.2 Command scripts

You can create command scripts in the following ways:

- Manually, using a text editor.
- By opening a log file during a debugging session.

You can also create a command script from a journal file. However, you must edit the contents of the file to make sure any lines that do not contain CLI commands are commented out or removed.

Comments in command scripts

You can use comments in your command scripts. Any characters identified as belonging to a comment are ignored by RealView Debugger. The following rules apply to comments in command scripts:

- C style comments begin with a slash followed by an asterisk /*) and end with an asterisk followed by a slash (*/. Also the comment text and the delimiters must be on a single line:
 - valid comment

```
/* comment */
```

These comments appear in log and journal files.

 - invalid comment

```
/*
    another comment
*/
```
- C++ style comments begin with two slashes //) and end when the end of the line is reached, for example:


```
// This is a line comment
// Copyright (c) ARM Limited
```

- Comments that begin with //, but are not placed after a command, do not appear in any log and journal files.
- Comments can begin with a semicolon (;), for example:
; A comment
- Comments can begin with //# and end when the end of the line is reached.
- Comments that begin with //#, but are not placed after a command, appear only in a journal file. Also, the //# prefix is replaced with ; in the that file.
- Only // or //# comments can be placed at the end of a command, for example:
ADD int value // integer value
- Comments cannot be nested.

15.1.3 Macros

You can define and use macros in a command script, in addition to CLI commands. Macros enable you to perform complex operations that are not possible with CLI commands alone.

See also

- *Using macros in command scripts* on page 15-8
- Chapter 16 *Using Macros for Debugging*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - Chapter 3 *RealView Debugger Predefined Macros*.

15.2 Changing output buffering behavior

In the current release of RealView Debugger, output to the log files and journal files is buffered. This means that all lines are not immediately flushed to the specified file. To change this, so that output to a file is unbuffered, set the JOULOG_UNBUF environment variable to any value.

See also:

- Chapter 15 *Debugging with Command Scripts*.

15.3 Creating a log file for use as a command script

You can create a command script by logging the commands generated by RealView Debugger and those you enter to a log file. You can create a new log file, or append the commands to an existing log file.

To create a log file for use as a command script, or to append to an existing log file:

1. Select **Tools → Logs and Journal → Log File...** to display the Select File to Log to dialog box where the file can be located.

————— Note —————

Additional menu options enable you to open journal and STDIO log files, if required.

2. Either:
 - specify the path name of the new log file
 - locate a file created previously.
 For example, C:\home\my_user_name\my_log.log.
3. Click **Save** to confirm the settings and close the dialog box.
If the specified log file already exists, RealView Debugger displays the File Exists prompt. This gives you the option to append or replace.
4. Click:
 - **Yes** to append new commands to those already saved in the file
 - **No** to replace, or overwrite, any commands already saved in the file
 - **Cancel** to close the prompt and discontinue the log file access.

Output is recorded automatically in the specified file, unless you choose to cancel logging.

RealView Debugger shows that it is recording using the status display area at the bottom of the Code window, shown in Figure 15-1.

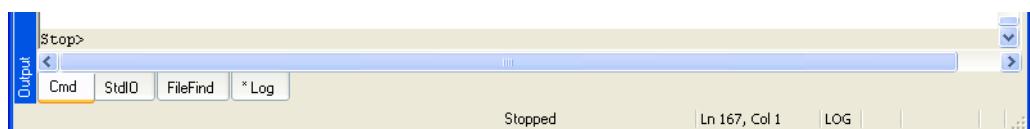


Figure 15-1 Status line indicating an open log file

See also:

- *Types of log and journal files* on page 1-44.

15.4 Creating log and journal files at start-up

You can start RealView Debugger and open a log or journal file for writing. Do this from a command prompt window, or create a desktop shortcut, for example:

```
rvdebug.exe --stdiolog "C:\rvd\test_files\STDIO_tst_file.log"
rvdebug.exe --jou "C:\rvd\test_files\Jou_tst_file.jou"
rvdebug.exe --log "C:\rvd\test_files\Log_tst_file.log"
```

Note

If the file does not exist, RealView Debugger creates it. Where the file exists, RealView Debugger overwrites the current contents, without displaying a warning message.

If the log and journal files are successfully opened, an indicator for each type is displayed in the status bar of the Code window, shown in Figure 15-2.

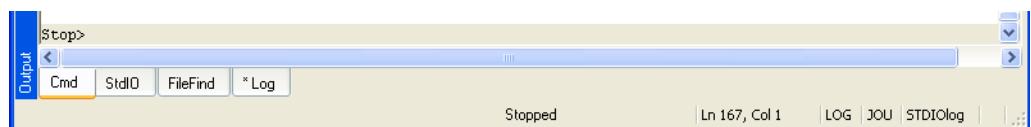


Figure 15-2 Status line indicating open log and journal files

When RealView Debugger starts to write to the log file, it records the filename as the first entry, for example:

```
;;;LOG FILE: C:\rvd\test_files\my_log.log
```

See also:

- *Starting RealView Debugger from the command line* on page 2-2.

15.5 Closing log and journal files

If you are recording a log, or journal file and you try to start a new recording, RealView Debugger gives you the option to close the current file so that a new file can be used.

To close a log or journal file:

1. Select **Tools** → **Logs and Journal** → **Close Logs/Journals...** from the Code window main menu to display the List Selection dialog box. This lists the log and journal files that you have opened. Figure 15-3 shows an example:

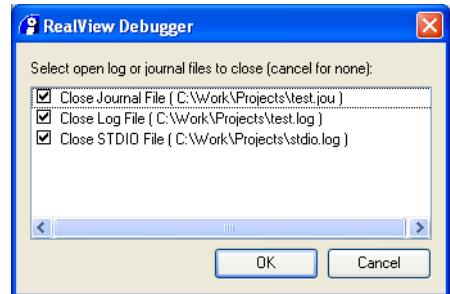


Figure 15-3 Close log and journal files selection box

2. Select the check box for each file that you want to close.
3. Either:
 - Click **OK** to close selected files and close the List Selection dialog box.
 - Click **Cancel** to leave all files open and close the List Selection dialog box.

Use a text editor of your choice to view the contents of your log and journal files. You can then edit the commands shown in a log file to create a command script.

See also:

- Chapter 15 *Debugging with Command Scripts*.

15.6 Using macros in command scripts

You can use macros in command scripts to perform complex operations that cannot be done with CLI commands alone.

Example 15-1 shows an example command script with a macro.

Example 15-1 Example command script with a macro

```
// The C-style escape character \\ is used in the FOPEN command and fopen()
// macro, because the path is in double quotes.
// You can use single quotes in the FOPEN command without the escape character,
// but you cannot use single quotes in the fopen() predefined macro.
FOPEN 200, "c:\\myfiles\\data.txt"
FPRINTF 200, "message printed to user-defined file"
VCLOSE 200

define /R void read_msg()
{
    char buffer[37];
    int nbytes;
    int retval;
    retval = fopen(250,"c:\\myfiles\\data.txt","r"); // open for read-only
    if (retval < 0)
        error(2,"Source file not opened!\n",101);
    else {
        nbytes = fread(buffer, 1, 21, 250);
        $FPRINTF 1, "%s\n", buffer$;           // Output to Code window
        fclose(250);                         // Close file
    }
}
.

// Run the macro
read_msg()
```

You can also use CLI commands within macros, if required. However, RealView Debugger prohibits the use of some commands in macros.

See also:

- *Creating a macro* on page 16-4
- *Using CLI commands in macros* on page 16-6
- *Loading user-defined macros* on page 16-9
- *Chapter 16 Using Macros for Debugging*.

15.7 Running command scripts

RealView Debugger enables you to run command scripts in the GUI or as part of the startup command.

Use a text editor to create a file of commands that can then be submitted to RealView Debugger to control a debugging session.

Note

Some commands require that you first connect to a target. You can either do this in the command script, or before you run the script.

See also:

- *Running scripts using the Scripts toolbar*
- *Running scripts using the Include Commands from File option* on page 15-10
- *Running scripts when starting RealView Debugger* on page 15-10
- *Referencing scripts from other command scripts* on page 15-10
- *Considerations when running command scripts* on page 15-11

15.7.1 Running scripts using the Scripts toolbar

To run a command script using the Scripts toolbar:

1.  Click **Add Script** in the Scripts toolbar.
The Choose Include File dialog box is displayed.
 2. Locate the required script file.
By default, RealView Debugger looks for *.inc and *.log files.
 3. Select the script file to be added to the Scripts toolbar.
You can select multiple scripts files to be added, if required.
 4. Click **Open** to load the command script.
The location of the each selected script file is added to list of script files in the toolbar.
- Figure 15-4 shows an example:

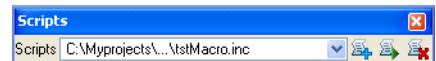


Figure 15-4 Script file added to Scripts toolbar

5.  Click **Run Script** to run the script.
6. To run a previously added command script:
 - a. Select the required script from the script drop-down list.
 - b. Click **Run Script**.

When a script is running a Script indicator is displayed in the Code window status bar.

If you want to remove a script from the script list:

1. Select the required script from the script drop-down list.
2.  Click **Remove Script**. The script entry is removed from the list. However, the script file is not deleted from your workstation.

15.7.2 Running scripts using the Include Commands from File option

To run a command script:

1. Select **Include Commands from File...** from the **Tools** menu to display the Select File to Include Commands from dialog box.
2. Locate the required script file.
By default, RealView Debugger looks for *.inc and *.log files.
3. Click **Open** to load and execute the command script.

When a script is running a Script indicator is displayed in the Code window status bar.

You can also use the INCLUDE command at the prompt in the **Cmd** tab to run a command script, for example:

```
None> include 'C:\Myprojects\scripts\tstMacro.inc'
```

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the INCLUDE command.

15.7.3 Running scripts when starting RealView Debugger

You can start RealView Debugger with a command script in the following ways:

- Start the RealView Debugger GUI and run the command script, for example:
`rvdebug.exe --inc "C:\Myprojects\scripts\tstMacro.inc"`
 When a script is running a Script indicator is displayed in the Code window status bar.
- Start RealView Debugger in batch mode and run the command script, for example:
`rvdebug.exe --batch --inc 'C:\Myprojects\scripts\tstMacro.inc'`

Note

If you use --batch without --inc, RealView Debugger displays an error message and exits.
 If you use only -inc, the script file is run with the GUI enabled.

Note

If the script does not connect and load an image, you must also specify the --init and --exec arguments to the rvdebug command.

See also

- *Starting RealView Debugger from the command line* on page 2-2.

15.7.4 Referencing scripts from other command scripts

You can use the INCLUDE command in a command script to pull in commands from other command scripts. You must use a single INCLUDE command for each file to be included, for example:

```
// CLI commands  
...  
include 'C:\Myprojects\scripts\script_1.inc'  
// more CLI commands  
...
```

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the INCLUDE command.

15.7.5 Considerations when running command scripts

When you run a command script, RealView Debugger sets the environment variable RVDEBUG_INCLUDE_BASE to the location of that command script. Therefore, you can use this environment variable in your script if required. The environment variable definition exists only for the current debugging session, and changes when you run additional command scripts.

The environment variable is not changed if you use the INCLUDE command in a command script.

To see the current value of the environment variable, enter the following CLI command:

```
host set RVDEBUG_INCLUDE_BASE
```

15.8 Creating a script that writes information to a user-defined window

Many CLI commands enable you to redirect the output to user-defined windows when running RealView Debugger in GUI mode.

To create a script that writes information to a user-defined window:

1. Create a script, and place the VOPEN command in your command script before any other commands that support window IDs.
2. For those commands that support window IDs, enter the ID of the window where the command output is to be directed.

You can also use the fwrite predefined macro to send output to this window.

For example:

```
VOPEN 100
FPRINTF 100, "message printed in user-defined window"
```

See also:

- *CLI commands that support user-defined window IDs.*

15.8.1 CLI commands that support user-defined window IDs

The following CLI commands support user-defined window IDs:

- BREAKACCESS
- BREAKEXECUTION
- BREAKINSTRUCTION
- BREAKREAD
- BREAKWRITE
- DCOMMANDS
- DLOADERR
- DOS_resource_list commands
- DPPIPEVIEW
- DTBOARD
- DTBREAK
- DTFILE
- DTRACE
- EXPAND
- FPRINTF
- REGINFO
- SHOW
- VCLOSE
- VMACRO.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

15.9 Creating a script that accesses a user-defined file

Many CLI commands enable you to redirect the output to user-defined files. In addition, you can read from and write to files using predefined macros.

To create a script that accesses a user-defined file:

1. Create a script, and place the VOPEN command in your command script before any other commands that support file IDs.
Alternatively, use the fopen predefined macro to open a file.
2. For those commands that support file IDs, enter the ID of the file where the command output is to be directed.
3. Place the VCLOSE command in your command script to close the file before the command script exits.

For example:

```
FOPEN 200, "c:\\myfiles\\data.txt"
FPRINTF 200, "message written to user-defined file"
VCLOSE 200
```

See also:

- *CLI commands that support user-defined window IDs* on page 15-12.
- *CLI commands that support user-defined file IDs*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

15.9.1 CLI commands that support user-defined file IDs

The following CLI commands support user-defined file IDs:

- BREAKACCESS
- BREAKEXECUTION
- BREAKINSTRUCTION
- BREAKREAD
- BREAKWRITE
- DCOMMANDS
- DLOADERR
- DOS_resource_list commands
- DPIPEVIEW
- DTBOARD
- DTBREAK
- DTFILE
- DTRACE
- EXPAND
- FOPEN
- FPRINTF
- REGINFO
- SHOW
- VMACRO.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

15.9.2 Predefined macros that support user-defined file IDs

The following predefined macros support user-defined file IDs:

- `fclose`
- `fgetc`
- `fopen`
- `fputc`
- `fread`
- `fwrite`.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical predefined macro reference* on page 3-6.

Chapter 16

Using Macros for Debugging

This chapter describes how to define macros for use during your debugging session, and how to save and edit your macros. It includes:

- *About using macros for debugging* on page 16-2
- *Creating a macro* on page 16-4
- *Loading user-defined macros* on page 16-9
- *Running a macro* on page 16-12
- *Editing a macro* on page 16-14
- *Copying a macro* on page 16-16
- *Viewing a macro* on page 16-17
- *Deleting a macro* on page 16-18
- *Using macros in combination with other commands* on page 16-19
- *Stopping execution of a macro* on page 16-22.

16.1 About using macros for debugging

In RealView® Debugger, a macro is a C-like function that is invoked by entering a single command using the macro name.

The following sections give an overview of macros in RealView Debugger:

- *What is a macro?*
- *Properties of macros* on page 16-3.

16.1.1 What is a macro?

Macros are:

- interpreted C code running on the host with access to target memory and symbols
- user-defined debugger symbols (in host or target memory)
- debugger functions.

Macros can access debugger variables, external windows and programs, and can be attached to breakpoints, aliases, and windows.

A macro can contain:

- a sequence of expressions
- string formatting controls
- statements
- calls to other macros
- predefined macros
- target functions
- debugger commands.

You can define and use macros at any time during a debugging session to use the commands or statements contained in the macro. You call the macro with a single command using the name. The macro definition might contain parameters that you change each time the macro is called.

When a macro is defined, you can use it as:

- a complex command or in an expression
- an attachment to the G0 command, or with the GOSTEP command
- an attachment to a breakpoint to create breakpoint condition testing
- an attachment to a user-defined window or file where the macro can send information.

See also

- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- *Displaying information in a user-defined window* on page 13-87
- *Saving information to a user-defined file* on page 13-90
- *Macro return values* on page 16-7
- *Using macros in combination with other commands* on page 16-19
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Macro definition* on page 1-10

- *Alphabetical command reference* on page 2-12 for details of the G0 and GOSTEP commands
- Chapter 3 *RealView Debugger Predefined Macros*.

16.1.2 Properties of macros

Macros can:

- have return values
- contain C expressions
- contain certain C statements
- have arguments
- define macro local variables
- use conditional statements
- call other macros and predefined macros
- be used in expressions, where they return values
- reference target variables and registers
- reference user-defined variables, in debugger or target memory
- execute most debugger CLI commands
- be defined in a debugger command script.

Macros cannot:

- be recursive
- define global variables
- define static variables
- define other macros.

16.2 Creating a macro

You can create a macro in the RealView Debugger GUI with the Add/Edit Macros dialog box, or in a command script. It is not possible to create macros directly on the command line.

The number of macros that can be defined is limited only by the available memory on your workstation.

— Note —

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. Disconnecting also clears any macros.

See also:

- *Creating a macro with the RealView Debugger GUI*
- *Creating a macro in a text file directly* on page 16-5
- *Using CLI commands in macros* on page 16-6
- *Macro return values* on page 16-7
- *Considerations when creating macros* on page 16-8.

16.2.1 Creating a macro with the RealView Debugger GUI

To create a macro for use with a debug target:

1. Connect to your target.
2. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box.

When you open this dialog box, the Existing Macros display list is empty, unless you have previously loaded macros into RealView Debugger.

The Macro Entry Area gives advice on how to use the buttons, **New**, **Show**, and **Copy**.

This area shows the definition of the macro when it has been created.

3. Click **New** to create the macro.

This inserts the default name `int Macro()` in the Name data entry field and inserts `{}` in the Macro Entry Area ready for editing.

4. Edit the default macro name so that it shows `int tutorial(var1)`.

— Note —

A user-defined macro must not have the same name as a predefined RealView Debugger macro.

5. Edit the macro contents to show:

```
int var1;
{
    $printf "value=%d",var1$;
    return var1;
}
```

When creating a macro, variables must be declared at the start of the macro definition. This also applies to macros that you create using a text editor.

Note

You must not include a period (.) to terminate the macro definition when using the Add/Edit Macros dialog box.

The Add/Edit Macros dialog box looks like the example shown in Figure 16-1.

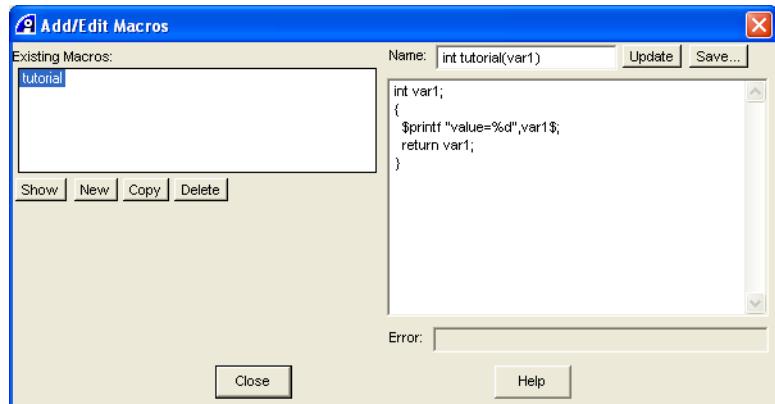


Figure 16-1 Creating a macro

The macro uses the PRINTF command and so the command must be enclosed by dollar signs (\$), shown in the Macro Entry Area.

6. Click **Update** to save the macro definition in the symbol table.
This adds the new macro to the Existing Macros display list. If there are any errors in the macro text, you are notified when the macro is saved.
7. Click **Save...** to display the Save Macro dialog box.
8. Locate the directory where you want to save the macro definition, and either:
 - select an existing file
 - enter the required filename.
9. Click **Save** to save the new macro, and close the Save Macro dialog box.
If the specified file already exists, RealView Debugger displays a prompt that gives you the option to append the macro to the existing file or to overwrite the existing contents:
 - click **Yes** to append the macro to the existing file
 - click **No** to overwrite the existing contents.
10. Click **Close** to close the Add/Edit Macros dialog box and return to the Code window.

See also

- *Connecting to a target* on page 3-27
- *Running a macro* on page 16-12.

16.2.2 Creating a macro in a text file directly

You can create one or more macros in a text file directly. Each macro definition must be terminated by a period (.), for example:

```
define /R return_type macro_1()
{
    // Code for macro_1
    ...
    return value; //optional statement
```

```

}

define /R return_type macro_2()
{
    // Code for macro_2
    ...
    return value; //optional statement
}
.
```

Note

A user-defined macro must not have the same name as a predefined RealView Debugger macro.

16.2.3 Using CLI commands in macros

You can define a macro that contains a sequence of debugger CLI commands. When used in this way, each command must be enclosed by dollar signs (\$). Example 16-1 shows an example.

Example 16-1 Using CLI commands in macros

```

some_commandsdefine int registers()
{
    int mIndex;
    for(mIndex = 0; mIndex < 6; mIndex++)
    {
        macro_bin_str[mIndex + 8] = macro_cpsr_mode[mIndex + (6*macro_cpsr_key)];
    }
    macro_bin_str[14] = 0x00;
    $printf " r0 = 0x%08x" ,@r0$;
    $printf " r1 = 0x%08x" ,@r1$;
    $printf " r2 = 0x%08x" ,@r2$;
}
.some_commands

```

Macros containing commands are similar to command files and can be used for setting up complex initialization conditions. These macros are executed by entering the macro name and any parameters on the RealView Debugger command line.

Because macros can return a value, they can also be used in expressions.

Using variable substitution in commands within a macro

You can substitute the value of an integer variable in a CLI command before the command is executed. A format specifier can also be included:

d decimal format

h or x hexadecimal format (this is the default).

The syntax for variable substitution is \${variable[:format]}. For example:

```

define /R int tstMacro()
{
    int num;
    num = 1;
    $FOPEN 150, "C:\\myfiles\\myfile${num:d}.txt"; // substitution
    $PRINTF 150, "Test value: %d", num$;

```

```
$VCLOSE 150$;
}
```

The filename in this example is `myfile1.txt`. The text written to the file is "Test value: 1".

Considerations for breakpoint and tracepoint commands

If you want to use breakpoint and tracepoint commands within a macro and use a variable to specify the address, then enclose the variable name between the characters \${ and }. Otherwise, when you view the breakpoint or tracepoint command in the Break/Tracepoints view, the variable name is shown instead of the address. For example, if `addr` has the value `0x9000`:

- `$breakinstruction addr$;`
This shows the breakpoint command as `breakinstruction addr`.
- `$breakinstruction ${addr}$;`
This shows the breakpoint command as `breakinstruction 0x9000`.
You can also use the optional format specifier, for example `${addr:d}`.

Commands prohibited inside a macro

RealView Debugger prohibits the use of the following commands inside a macro:

- ADD
- BOARD
- CONNECT
- DEFINE (unless it is the macro definition itself)
- DELETE
- DISCONNECT
- GOSTEP
- HELP
- HOST
- INCLUDE
- QUIT.

Note

Macros that are not directly invoked from the command line cannot use execution-type commands, such as GO or STEPINSTR.

See also

- *Macro return values*
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12
 - Chapter 3 *RealView Debugger Predefined Macros*.

16.2.4 Macro return values

The type of the macro return value is specified by `return_type` when you define the macro, and can be one of the following:

- `char`
- `double`

- `float`
- `int`
- `long`
- `short`
- `unsigned`.

If `return_type` is not specified, then type `int` is assumed.

The macro return value determines the execution behavior when used with the following commands:

- breakpoint commands
- `BGLOBAL` command
- `G0` command
- `GOSTEP` command.

See also

- *Controlling breakpoint behavior with macro return values* on page 12-23
- *Attaching a macro to the GO command* on page 16-20
- *Using a macro when stepping* on page 16-20
- *Attaching a macro to a breakpoint* on page 16-21
- *Attaching a macro to a global breakpoint* on page 16-20
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Macro definition* on page 1-10
 - *Alphabetical command reference* on page 2-12 for details of the breakpoint commands and the `BGLOBAL`, `G0` and `GOSTEP` commands.
 - Chapter 3 *RealView Debugger Predefined Macros*.

16.2.5 Considerations when creating macros

Be aware of the following when creating macros:

- A user-defined macro must not have the same name as a predefined RealView Debugger macro.
- Some CLI commands are prohibited inside a macro.
- The `/R` switch in the macro definition makes sure that the macro is replaced when it is reloaded.

If you use the same name as a predefined macro or omit the switch, RealView Debugger outputs the error message:

Error: E004D: Symbol with this name already exists.

See also

- *Commands prohibited inside a macro* on page 16-7
- *Reloading a user-defined macro* on page 16-10.

16.3 Loading user-defined macros

Before you can use a macro, you must load it into RealView Debugger.

Note

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. Disconnecting also clears any macros.

See also:

- *Loading a user-defined macro using the Scripts toolbar*
- *Reloading a user-defined macro* on page 16-10
- *Reloading a user-defined macro* on page 16-10
- *Loading macros on connection* on page 16-10.

16.3.1 Loading a user-defined macro using the Scripts toolbar

To load a user-defined macro using the Scripts toolbar:



1. Click **Add Script** in the Scripts toolbar.

The Choose Include File dialog box is displayed.

2. Locate the script file containing your macro definition.

By default, RealView Debugger looks for *.inc and *.log files.

3. Click **Open** to load the script file.

The location of the script file is added to the top of the list of script files in the toolbar, shown in Figure 16-2.



Figure 16-2 Script file added to Scripts toolbar



4. Click **Run Script** to run the script.

5. To run a previously added script:

- a. Select the required script from the script drop-down list.

- b. Click **Run Script**.

If you want to remove a script from the script list:



1. Select the required script from the script drop-down list.

2. Click **Remove Script**. The script entry is removed from the list. However, the script file is not deleted from your workstation.

16.3.2 Loading a user-defined macro using the Include Commands from File option

To load a user-defined macro:

1. Select **Include Commands from File...**, from the **Tools** menu to display the Select File to Include Commands from dialog box.

2. Locate the file containing your macros.

By default, RealView Debugger looks for a .inc or a .log file.

3. Click **Open** to load the macros defined in the script file.

You can also use the INCLUDE command at the command prompt to run a command script, for example:

```
None> include 'C:\Myprojects\scripts\tstMacro.inc'
```

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the INCLUDE command.

16.3.3 Reloading a user-defined macro

To reload a macro into RealView Debugger:

1. Select **Include Commands from File...** from the **Tools** menu to display the Select File to Include Commands from dialog box.
2. Highlight the required .inc file and then click **Open**. This loads the selected macro into RealView Debugger.
If there is an error in the .inc file, an error message is generated in the Output view and the macro is undefined.
3. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box, where your macro is shown in the Existing Macros display list.

You can load several macros in this way ready for use in your debugging session.

Note

If a macro definition does not include the /R switch, then the following error message is displayed:

```
Error: E004D: Symbol with this name already exists.
```

See also

- *Creating a macro in a text file directly* on page 16-5.

16.3.4 Loading macros on connection

You can load one or more macros automatically when you connect to a given target. The macros must be defined in a script file, which can be loaded on connection.

To load one or more script files containing macro definitions on connection:

1. Select **Connect to Target** from the **Target** menu. The Connect to Target window is displayed.
2. Select **Configuration** from the Grouped By drop-down list.
3. Expand the Debug Interface containing the Debug Configuration to be customized.
4. Make sure that there are no targets connected on the Debug Configuration.

Note

You cannot customize a Debug Configuration when the debugger is connected to a target in that Debug Configuration.

5. Right-click on the required Debug Configuration to display the context menu.
 6. Select **Properties...** from the context menu. The Connection Properties dialog box is displayed.
 7. Click the **Commands** tab to display the commands list.
 8. To add a command:
 - a. Click the **Browse** button to display the Choose Command file dialog box.
 - b. Locate the required script file.
 - c. Click **Open**. The Choose Command file dialog box closes, and the appropriate include command is added to the command list.
 - d. Repeat these steps to add more commands as required.
- Alternatively:
- a. Enter the command in the command entry field. For example, enter:
`include C:\rvd\Test_files\tutorial.inc`
- Note**
- If the path contains spaces, then you must delimit the path with single quotes.
- b. Click the **Add** button.
The command is added to the command list.
9. Click **OK** to save your changes.

When you connect to a target in the Debug Configuration, the commands are executed in the order listed in the command list. If RealView Debugger cannot locate one of the specified files, it displays a message box and gives you the option to abort the connection.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *Changing the order of settings that have multiple values* on page 3-12
 - *Running CLI commands automatically on connection* on page 3-41.

16.4 Running a macro

You can run a macro from:

- the RealView Debugger command line, either directly or in a command script
- another macro definition (but a macro cannot be called recursively).

See also:

- *Running a macro directly at the RealView Debugger command line*
- *Running a macro that returns a value*
- *Running a macro that has the same name as a CLI command*
- *Running a macro that has the same name as a target function* on page 16-13.

16.4.1 Running a macro directly at the RealView Debugger command line

You enter the macro name followed by a set of parentheses containing any macro arguments separated by commas. For example:

```
tstMacro()  
tstMacroArgs(arg1,arg2,arg3)
```

Macro names are case sensitive and must be entered as shown in the definition. Any macro arguments are converted to the types specified in the macro definition. If RealView Debugger cannot convert the arguments it generates an error message.

16.4.2 Running a macro that returns a value

If a macro returns a value, then use the CEXPRESSION command to run the macro. This displays the value returned, for example:

```
> cexpression tutorial(4)  
value=4 Result is: 4 0x00000004
```

— Note —

The macro tutorial() is the macro created in *Creating a macro* on page 16-4.

16.4.3 Running a macro that has the same name as a CLI command

You can define a macro with a name that is identical to a CLI command used by RealView Debugger. However, because you must include the parentheses after the macro name to run it, RealView Debugger is able to distinguish the macro from the CLI command.

For example, if you have a macro called memmap(), then:

- memmap() runs the macro
- memmap runs the MEMMAP command.

You can also run a macro by using it as an argument to the MACRO command, for example:

```
MACRO memmap()
```

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the MACRO command.

16.4.4 Running a macro that has the same name as a target function

Macros take higher precedence than target functions. If a target function and a macro have the same name, the macro is executed unless the target function is fully qualified.

For example, `strncpy` is a predefined debugger macro, while `PROG\strncpy` is a function within the module PROG. The predefined macro is referenced as `strncpy(dest,)`, while `PROG\strncpy(dest,)` refers to the function within PROG.

16.5 Editing a macro

You can edit a macro definition to fix any problems in the macro or modify the behavior of the macro.

To edit a macro that you have created:

1. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box.
2. Select the macro to be edited from the Existing Macros display list.
3. Click **Show** to see the contents of the macro.
4. Change the macro name and arguments, if required.
5. In the Macro Entry Area, change the macro definition as required.
6. Click **Update** to update the macro definition in the symbol table.

————— **Note** —————

The updated macro is moved to the bottom of the Existing Macros list.

7. Click **Save...** to save the updated macro in the same location.
This generates a prompt to enable you to Append or Replace the existing file.
8. Click **No** to replace the existing script file.
9. Click **Close** to close the Add/Edit Macros dialog box.

————— **Note** —————

Remember to save your macros before you exit RealView Debugger. There is no warning if any macro definitions are unsaved when you exit RealView Debugger.

See also:

- *Editing the example tutorial() macro* on page 16-15.

16.5.1 Editing the example tutorial() macro

To edit the `tutorial()` macro:

1. Make sure you have:
 - a. Created the `tutorial()` macro as described in *Creating a macro* on page 16-4.
 - b. Loaded the macro.
2. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box.
3. Select **tutorial** from the Existing Macros display list.
4. Click **Show** to see the contents of the macro.
5. Change the macro name and arguments to:

```
/R int tutorial(var1,var2,var3)
```

————— **Note** —————

The `/R` (replace) switch must be included before the macro name because the macro is already defined in the symbol table.

-
6. In the Macro Entry Area, change the macro definition to:


```
int var1;
int var2;
int var3;
{
    $printf "value of var1=%d",var1$;
    $printf "value of var2=%d",var2$;
    $printf "value of var3=%d",var3$;
    return var1+var2+var3;
}
```
 7. Click **Update** to update the macro definition in the symbol table.
 8. Click **Save...** to save the updated macro in the same file containing the original definition of `tutorial()`.
- This generates a prompt to enable you to Append or Replace the existing file.
9. Click **No** to replace the macro definition in the existing script file.
 10. Click **Close** to close the Add/Edit Macros dialog box.

Testing the macro

To test the macro, enter:

```
> cexpression tutorial(1,2,3)
value of var1=1
value of var2=2
value of var3=3
Result is: 6 0x00000006
```

See also

- *Loading user-defined macros* on page 16-9.

16.6 Copying a macro

You can copy a macro if you want to create a new macro based on an existing macro.

The example in this procedure assumes that you have modified the `tutorial()` macro as described in *Editing the example tutorial() macro* on page 16-15.

You can use an existing macro to form the basis of a new macro:

1. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box. The Existing Macros display list shows any macro you have previously created.
2. Select the macro to be edited from the Existing Macros display list.
3. Click **Show** to see the contents of the macro.
4. Click **Copy**.

This automatically adds an integer number to the end of the macro name in the Name field, starting at one. You can change the name, if required.

For the macro created in *Editing the example tutorial() macro* on page 16-15, the name of the new macro becomes `tutorial1(var1,var2,var3)`. Subsequent copies are numbered sequentially, for example, `tutorial2(var1,var2,var3)` and `tutorial3(var1,var2,var3)`.

5. In the Macro Entry Area change the body of the macro as required.
6. Click **Update** to define the new macro definition in the symbol table.

————— **Note** —————

The updated macro is moved to the bottom of the Existing Macros list.

7. View the Output view message, assuming that you do not change the default name:
`def int tutorial1(var1,var2,var3)`
8. Click **Save** to save the updated macro in the usual way.
9. Click **Close** to close the Add/Edit Macros dialog box.

16.7 Viewing a macro

View the contents of a macro by:

- Opening the script file defining the macro in RealView Debugger.
- Opening the script file in an external text editor.
- Using the Add/Edit Macros dialog box after the macro is loaded:
 1. Load the macro.
 2. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box. The Existing Macros display list shows any macro you have previously loaded.
 3. Select the required macro in the Existing Macros list.
 4. Click **Show** to see the contents of the macro.

When viewing the macro contents in the Add/Edit Macros dialog box, the Macro Entry Area in the dialog box does not show the macro **DEFINE** command or the terminator (a period used as the first and only character on the last line).

- Using the **SHOW** command after the macro is loaded:
 1. Load the macro.
 2. Specify the macro as an argument to the **SHOW** command. For example:


```
> show tutorial
def int tutorial(var1,var2,var3)
int var1;
int var2;
int var3;
{
    $printf "value of var1=%d",var1$;
    $printf "value of var2=%d",var2$;
    $printf "value of var3=%d",var3$;
    return var1+var2+var3;
}
```

When viewing a macro with the **SHOW** command, the terminator is not displayed.

————— **Note** —————

You cannot use the **SHOW** command to view predefined macro definitions.

See also:

- *Loading user-defined macros* on page 16-9.
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the **SHOW** command.

16.8 Deleting a macro

Deleting a macro clears the macro definition from your RealView Debugger session.

To delete a macro definition from the current debugging session:

1. Select **Add/Edit Debugger Macros...** from the **Tools** menu to display the Add/Edit Macros dialog box. The Existing Macros display list shows any macro you have previously created.
 2. Select the macro to be deleted from the Existing Macros display list.
 3. Click **Delete** to see delete the macro.
- The macro is deleted. This does not delete any file containing the macro. You can only delete one macro at a time in this way.
4. Click **Close** to close the Add/Edit Macros dialog box.

————— **Note** —————

You can also delete a macro, and all associated symbols, using the **DELETE** command.

See also:

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the **DELETE** command.

16.9 Using macros in combination with other commands

You can run macros directly through the command line, or in a command script. However, you can also use macros in combination with other commands. The procedures described in this section use a macro called `my_macro()` as an example.

— Note —

There are limitations on the commands that you can use in a macro when attaching them to commands.

See also:

- *Using an alias to run a macro*
- *Redirecting macro output to a user-defined window or file*
- *Attaching a macro to the GO command* on page 16-20
- *Using a macro when stepping* on page 16-20
- *Attaching a macro to a global breakpoint* on page 16-20
- *Attaching a macro to a breakpoint* on page 16-21
- *Limitations of using macros with other entities* on page 16-21.

16.9.1 Using an alias to run a macro

To use an alias to run a macro, use the ALIAS command, for example:

```
Stop> ALIAS my_alias=my_macro()
Stop> my_alias
```

— Note —

You cannot use an alias to run a macro as part of another command, such as a breakpoint command.

See also

- *Commands prohibited inside a macro* on page 16-7
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the ALIAS command.

16.9.2 Redirecting macro output to a user-defined window or file

To redirect output from a macro to a user-defined window or file, use the VMACRO command, for example:

```
VMACRO 250, my_macro()
```

See also

- *Commands prohibited inside a macro* on page 16-7
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the VMACRO command.

16.9.3 Attaching a macro to the GO command

To invoke a macro that controls the action of the GO command enter, for example:

```
GO \DHY_1\#149; my_macro()
```

In this example, the program runs until the temporary breakpoint at line 149 in dhry_1.c is hit, which causes the macro to run. The return value from the macro determines whether the execution continues or stops.

See also

- *Commands prohibited inside a macro* on page 16-7
- *Macro return values* on page 16-7
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the GO command.

16.9.4 Using a macro when stepping

To invoke a macro when stepping, use the GOSTEP command, for example:

```
GOSTEP my_macro()
```

Be aware that if the macro returns zero, then execution stops after each step. Click **Run** to continue.

See also

- *Commands prohibited inside a macro* on page 16-7
- *Macro return values* on page 16-7
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the GOSTEP command.

16.9.5 Attaching a macro to a global breakpoint

To invoke a macro as an action associated with a global breakpoint, use the BGLOBAL command, for example:

```
BGLOBAL,enable IRQ ; my_macro()
```

See also

- *Commands prohibited inside a macro* on page 16-7
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the BGLOBAL command.

16.9.6 Attaching a macro to a breakpoint

To invoke a macro as an action associated with a breakpoint, attach the macro either as a command qualifier or as an argument, for example:

- BREAKINSTRUCTION,macro:{my_macro()} 0x8100
- BREAKINSTRUCTION 0x8100; my_macro().

See also

- *Commands prohibited inside a macro* on page 16-7
- *Setting a breakpoint that depends on the result of a macro* on page 12-21
- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the breakpoint commands.

16.9.7 Limitations of using macros with other entities

If you use a macro with other entities, then execution-type commands cannot be used inside the macro. That is:

- GO
- GOSTEP
- STEPLINE, STEPO
- STEPINSTR, STEPOINSTR.

See also

- *Commands prohibited inside a macro* on page 16-7.

16.10 Stopping execution of a macro

When macros are run as commands they are queued for execution like any other debugger command when your program is executing. To stop execution of a macro, do one of the following:

- Select **Cancel Current Command** from the **Debug** menu to cancel the last command entered onto the queue. This can be used to stop any macro that is running. This does not take effect until the previous command has completed and so any effects might be delayed.
- Click **Stop** to stop a macro that is attached to a breakpoint.



Chapter 17

Configuring Workspace Settings

This chapter explains how to use workspaces in RealView® Debugger, and describes how to configure your workspace settings. It includes:

- *About workspace settings* on page 17-2
- *Initializing the workspace* on page 17-3
- *Opening workspaces* on page 17-4
- *Closing workspaces* on page 17-6
- *Creating an empty workspace* on page 17-8
- *Saving workspaces* on page 17-9
- *Viewing workspace settings* on page 17-11
- *Configuring workspace settings* on page 17-15.

Read this chapter in conjunction with Appendix A *Workspace Settings Reference* that contains a detailed description of the workspace settings.

17.1 About workspace settings

RealView Debugger uses a workspace to define:

- connection information
- debugger behavior
- windows (sizes and positions) and their attachment
- window contents and views
- user-defined editor settings and view options.

RealView Debugger uses a default workspace. Using a workspace enables you to maintain persistence between debugging sessions.

You can save a workspace configuration to a different file, if required. For example, you might want to have different views available in the RealView Debugger Code window, depending on the targets you are debugging. By setting up the Code window and saving the workspace, you do not have to manually set up the Code window in subsequent debugging sessions. You only have to open the saved workspace.

Working without a workspace might be useful to debug an executable file from another developer or for compatibility with other tools. This means that some persistence details are not available. If you close the current workspace, RealView Debugger uses the global configuration options, which contains a subset of the settings that are in a workspace.

There are descriptions of the general layout and controls of the RealView Debugger settings windows in the RealView Debugger online help topic *Changing Settings*. This chapter assumes familiarity with the procedures documented in that topic.

17.2 Initializing the workspace

The first time you run RealView Debugger after installation, it creates a default workspace to define your initial working environment. Two files are created in your RealView Debugger home directory to store settings:

- **rvdebug.aws** Contains workspace-specific settings that apply to the current workspace.
- **rvdebug.ini** Contains global configuration options that apply to all workspaces, or are used when working without a workspace.

By default, at the end of your session, the .aws file is updated to save the current workspace and is used when you start your next session. The global configuration file is updated when it is edited or at the end of your session if you are working without a workspace.

See also:

- *Examples of workspace start-up options.*

17.2.1 Examples of workspace start-up options

You can start RealView Debugger with a specified workspace from the command line, or by using a desktop shortcut, for example:

```
rvdebug.exe --aws="D:\RVD_home\my_user_name\myws_rvdebug.aws"
```

You can start a debugging session without a workspace, for example:

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

17.3 Opening workspaces

The workspace enables you to configure your debugging environment, which you can save. Therefore, you can change your debugging environment by opening previously saved workspace files.

See also:

- *Opening a workspace*
- *Opening a workspace from the Recent Workspaces list*
- *Using the same workspace on startup*
- *Considerations when opening workspaces* on page 17-5.

17.3.1 Opening a workspace

To open a workspace file that you have previously created:

1. Select **File** → **Workspace** → **Open Workspace...** from the Code window main menu to display the Select Workspace file to Open dialog box.
2. Locate the directory containing your saved workspace files.
3. Select the filename for the workspace file that you want to open.
4. Click **Open**. The current workspace is closed and the new workspace is loaded.

17.3.2 Opening a workspace from the Recent Workspaces list

When you open a workspace, the workspace file is added to the Recent Workspaces list. To open a workspace file from this list:

1. Select **File** → **Workspace** → **Recent Workspaces** from the Code window main menu to display the Recent Workspaces menu.
2. Select the workspace file to open. The current workspace is closed, and the selected workspace is loaded.

17.3.3 Using the same workspace on startup

To use the current workspace the next time that you start RealView Debugger, select **File** → **Workspace** → **Same Workspace on Startup** from the Code window main menu.

Note

This option is selected by default.

If selected, the current workspace path name is saved in your start-up file, which has the extension .sav, so that the same workspace is used at the next start-up.

Note

The default workspace is in rvdebug.sav in your home directory.

You can deselect this option so that the current workspace is not opened by default when you next start RealView Debugger. Unless you specify a workspace on the command line, RealView Debugger then runs without a workspace.

See also

- *Saving workspace settings on exit* on page 17-10.

17.3.4 Considerations when opening workspaces

Be aware of the following when opening workspaces:

- If you open a new workspace, RealView Debugger adds the objects specified in the new workspace to all existing objects. This usually means that at least one more Code window opens on your desktop.
- If you open a new workspace, you might see many new Code windows on your desktop. To avoid this, close any extra open Code windows before opening the new workspace.
- When you open a new workspace, it might contain settings that override the current configuration. Where there is a conflict, a warning message is displayed and the new workspace settings are used.

See also

- *Closing workspaces* on page 17-6.

17.4 Closing workspaces

When you close a workspace, RealView Debugger enables you to close any open objects. This might be useful to restore a clean desktop for the session, or before you open a new workspace.

To close your current workspace:

1. Select **File → Workspace → Close Workspace** from the Code window main menu.

The workspace is closed. In addition, if any connections were established, or additional Code windows displayed, a Close Open Objects selection dialog box is displayed, shown in Figure 17-1.

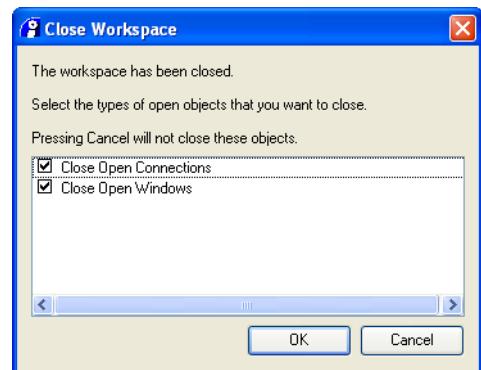


Figure 17-1 Close Open Objects selection dialog box

The display list shows the open objects, that is:

- connections to debug targets
- any Code windows that are open in addition to the default Code window.

Each entry has an associated check box that is selected by default.

————— **Note** —————

The Close Open Objects selection dialog box is not displayed if there are no open objects.

2. Either:

- Deselect the check box for each object that you do not want to close, then click **OK** to close the selected objects. The List Selection dialog box also closes.
- Click **Cancel** to close the List Selection dialog box without closing any selected objects.

See also:

- *Considerations when closing workspaces.*

17.4.1 Considerations when closing workspaces

If you close your current workspace, the following applies:

- The contents of the default Code window do not change, unless you choose to close open connections.
- If there are open objects, these do not change.
- Any open objects are saved in the workspace before it closes so that they can be re-used when it next opens.
- If there are no open objects, the current workspace closes immediately.

- If the Workspace Options window is open, this closes automatically before the current workspace closes. If you have changed any workspace settings, these are saved.
- If the Options window is open, this is not affected when the workspace closes.

17.5 Creating an empty workspace

You can create a blank workspace settings file at any point during your debugging session.

To create an empty workspace:

1. Select **File** → **Workspace** → **New Workspace...** from the Code window main menu to display the Select Filename for new Workspace dialog box.
2. Locate the directory where you want to save the new workspace file.
3. Enter a filename for the new workspace, for example `New_workspace.aws`. This becomes the current workspace.
4. Click **Save** to create the new workspace file.

The Select Filename for new Workspace dialog box closes.

The current workspace is closed, and the new workspace opens ready for you to use. This does not override the current configuration. That is, any open objects, such as connections and windows, remain open.

You must save settings to this new workspace settings file if you want it to be available when you next start RealView Debugger.

See also:

- *Saving workspaces* on page 17-9.

17.6 Saving workspaces

You can save the current workspace configuration at any time, either to the currently opened workspace file, or to a new workspace file.

See also:

- *Saving the current workspace settings*
- *Saving the current workspace settings to a new file*
- *Saving workspace settings on exit* on page 17-10.

17.6.1 Saving the current workspace settings

To save the current workspace settings, select **File** → **Workspace** → **Save Workspace** from the Code window main menu.

Any open objects, such as connections and windows, are also saved. These open objects are restored when you next open the workspace file.

See also

- *CONNECTION* on page A-14
- *WINDOW* on page A-15.

17.6.2 Saving the current workspace settings to a new file

To save the current workspace settings to a new file:

1. Select **File** → **Workspace** → **Save As Workspace...** from the Code window main menu to display the Select Workspace Name to Save-As dialog box.
2. Locate the directory where you want to save the workspace file.
3. Enter a new filename for the workspace.
4. Click **Save**.

The current workspace configuration is saved in the new workspace file. Any open objects, such as connections and windows, are also saved. These open objects are restored when you next open the workspace file.

See also

- *CONNECTION* on page A-14
- *WINDOW* on page A-15.

17.6.3 Saving workspace settings on exit

To save your workspace settings automatically when you exit RealView Debugger, select **File → Workspace → Save Settings on Exit** from the Code window menu.

— Note —

This option is selected by default.

This enables you to start your next debugging session in the same state.

If selected, settings are saved in your start-up file, which has the .sav extension by default, for use next time and the current workspace file is updated.

See also

- *Using the same workspace on startup* on page 17-4.

17.7 Viewing workspace settings

RealView Debugger provides:

- The Workspace Options window to enable you to examine, and change, workspace settings. Use this window to see the contents of the current workspace .aws file.
- The Options window to enable you to examine, and change, your global configuration settings. Use this window to see the contents of the rvdebug.ini file.

See also:

- *Viewing the current workspace settings*
- *Viewing the global configuration settings* on page 17-12
- *Workspace file references* on page 17-13
- *List of Entries pane* on page 17-13
- *Settings Values pane* on page 17-14.

17.7.1 Viewing the current workspace settings

To access your current workspace settings, select **File → Workspace → Workspace Options...** from the Code window main menu.

Note

The **Workspace Options...** menu option is unavailable if you have closed the workspace.

The Workspace Options window is displayed, shown in Figure 17-2, where you can view the current workspace settings or make changes.

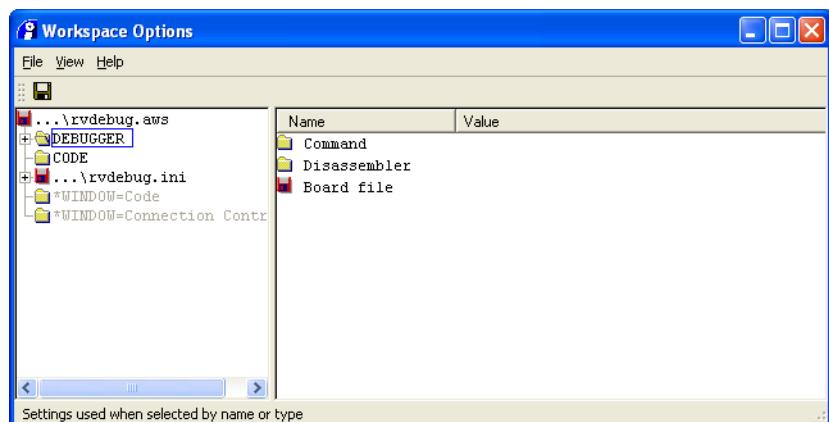


Figure 17-2 Workspace Options window

The Workspace Options window enables you to examine your current workspace settings and edit these settings to change the workspace or to create your own workspace files. The first time RealView Debugger opens the default workspace file, rvdebug.aws, the Workspace Options window contains only the start-up settings.

The left pane is the List of Entries pane, and shows configuration entries as a hierarchical tree with node controls.

The right pane is the Settings Values pane, and shows the settings for the group selected in the List of Entries pane.

When you close down RealView Debugger, your workspace settings file is updated with the current configuration, such as connections and open windows.

See also

- *List of Entries pane* on page 17-13
- *Settings Values pane* on page 17-14
- Appendix A *Workspace Settings Reference*.

17.7.2 Viewing the global configuration settings

To access your global configuration options, select **Options...** from the **Tools** menu.

The Options window is displayed, shown in Figure 17-3, where you can view the global configuration options used by the current workspace, or make changes.

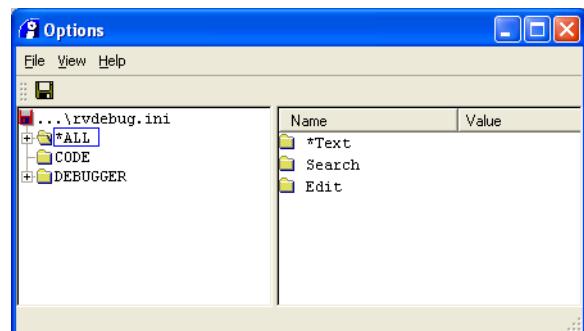


Figure 17-3 Options window

The Options window enables you to examine, and change, your current global configuration options. These settings are saved in the file rvdebug.ini and are included when the default workspace opens for the first time.

The left pane is the List of Entries pane, and shows configuration entries as a hierarchical tree with node controls.

The right pane is the Settings Values pane, and shows the settings for the group selected in the List of Entries pane.

If you are working without a workspace, use the Options window to make the changes.

See also

- *List of Entries pane* on page 17-13
- *Settings Values pane* on page 17-14
- Appendix A *Workspace Settings Reference*.

17.7.3 Workspace file references

The workspace and global configuration options contain the following file references:

Workspace file

This is the current workspace settings file.

Select this entry to see the full path name in the Description field at the bottom of the Workspace Options window.

Global configuration file

This is the global configuration file, `rvdebug.ini`, included in the current workspace.

You can set up DEBUGGER, CODE, or ALL groups in your workspace settings file or in your global configuration file. RealView Debugger issues a warning if conflicts are detected when the workspace opens and uses settings from the new workspace file.

17.7.4 List of Entries pane

The left pane of the Workspace Options window, the List of Entries pane, shows workspace entries as a hierarchical tree with node controls. Figure 17-4 shows an example.

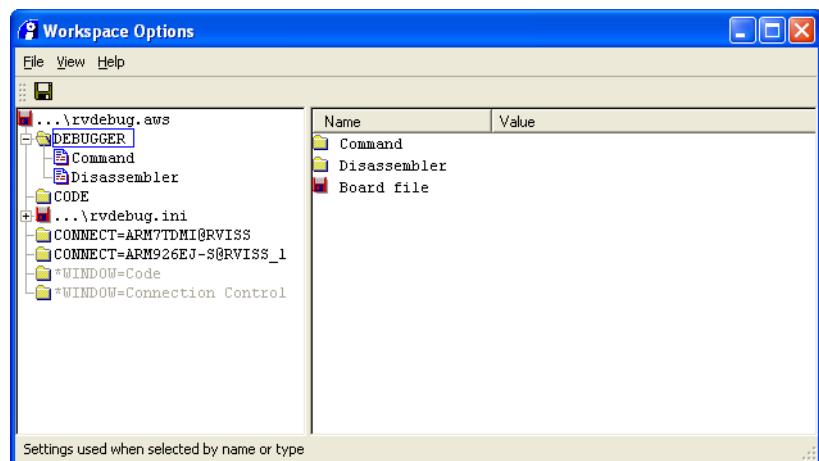


Figure 17-4 Example Workspace Options

Groups of settings are associated with an icon to explain their function:



Red disk This is a container disk file.

RealView Debugger uses this, for example, to specify an include file.



Yellow folder

This is a parent group containing other groups (*rules pages*) and/or entries.



Rules page A rules page is a container for settings values that you can change in the right pane. When deselected, the pencil disappears (see Figure 17-4).

This icon only appears in the left pane. Rules pages have a yellow folder icon in the right pane.

An asterisk (*) is placed at the front of an entry to show that it has changed from the default or was created by RealView Debugger. Figure 17-4 shows an example settings file that specifies a multiprocessor debugging session.

If you click on an entry in the left pane, a red box is drawn around it and the Description field is updated. At the same time, the right pane, the Settings Values pane, is updated to show the contents of the highlighted group.

See also

- *Settings Values pane*.

17.7.5 Settings Values pane

When you select on an entry in the left pane, shown in Figure 17-4 on page 17-13, a red box is drawn around it and the Description field is updated. At the same time, the right pane, the Settings Values pane, is updated to show the contents of the highlighted group.

Groups of settings are associated with an icon to explain their function:



Red disk

This specifies a disk file.

RealView Debugger uses this, for example, to specify a board file.



Yellow folder

This is a parent group or rules page containing other groups (*rules pages*) and/or entries.



String value

This is a text string. If more than one value can be assigned to the setting, a new setting is created with the chosen value, and is colored blue. The original setting remains available for you to add other values if required.



Numerical value

This is a numerical value.



True/False value

This has a value that is either True or False. To change the value, either:

- click on the switch button  in the Value field
- right-click on the setting, and select True or False from the context menu.



Preset selections value

This enables you to select the value from a list that is defined by RealView Debugger. If more than one value can be assigned to the setting, a new setting is created with the chosen value, and is colored blue. The original setting remains available for you to add other values if required.

An asterisk (*) is placed at the front of a setting to show that it has changed from the default or that it was changed by RealView Debugger.

See also

- *List of Entries pane* on page 17-13.

17.8 Configuring workspace settings

The following sections describe how to modify the workspace settings and restore the factory settings:

- *Restoring the factory settings*
- *Changing settings*
- *Resetting entries* on page 17-16
- *Considerations when configuring workspace settings* on page 17-16.

17.8.1 Restoring the factory settings

If you have made changes to your workspace settings file, or your global configuration file, you might want to restore the factory settings.

To restore the factory settings:

1. Exit RealView Debugger.
2. Locate the .aws and the .ini files in your RealView Debugger home directory.
3. Delete both files.
4. Start RealView Debugger and accept the option to create a new .aws file.

17.8.2 Changing settings

To change settings:

1. Select **File** → **Workspace** → **Workspace Options...** from the Code window main menu to display the Workspace Options window.
2. Make the required changes to the settings. These take effect in the current workspace only.
3. Select **Save Changes** from the Workspace Options window **File** menu to save our changes.
4. Select **New Code Window** from the Code window **View** menu to display a new Code window to see the effect. Close the new Code window when you have finished with it.
5. Select **Close Window** from the Workspace Options window **File** menu to close the Workspace Options window.

Setting up debugger options

To change the height of the Output view:

1. Expand the **DEBUGGER** group.
2. Select the **Command** group.
3. Right-click on the default **Num_lines** setting.
4. Select **Edit Value** from the context menu.
5. Use in-place editing to set the value to 10.
6. Press Enter to confirm your setting.
7. Save the updated version of the workspace settings file.

Note

To restore the Code window, select the option **Reset to Empty**.

See also

- *Viewing workspace settings* on page 17-11.

17.8.3 Resetting entries

An asterisk is placed at the front of any entry that you edit in the workspace settings file to show that it has changed from the default. This also applies to entries maintained by RealView Debugger.

To refresh all values, select **Refresh** from the **File** menu. This updates the window with the settings currently saved on disk. You are warned that any changes made since you last saved are lost.

When you have changed a value, right-click to see the context menu showing the option **Reset to Default**. Select this to change the value to the default and cancel any changes.

Right-click on a group of settings and select **Delete Contents** from the context menu to reset it back to empty. This deletes all the changed settings and restores the defaults. There is no undo.

17.8.4 Considerations when configuring workspace settings

The following notes apply to changing your workspace settings:

- Settings are applied in the order they are shown in the settings hierarchy in the left pane. This means that settings in the workspace file take priority over global configuration settings if a conflict arises when you open a workspace.
- If you edit the workspace settings, the .aws file is updated when you save the change. This change takes effect in any new Code windows you open in the current session.
- Use the Options window to make changes to global configuration options saved in the rvdebug.ini file.
- If you edit the global configuration options, the .ini file is updated when you save the workspace file. This change takes effect when the workspace next opens.
- Do not change the same setting in the Workspace Options window and the Options window at the same time because the views might not be consistent.

Appendix A

Workspace Settings Reference

This appendix contains reference details about settings that define the RealView® Debugger workspace and global configuration options. It contains the following sections:

- *DEBUGGER* on page A-2
- *CODE* on page A-6
- *ALL* on page A-7
- *CONNECTION* on page A-14.
- *WINDOW* on page A-15.

A.1 DEBUGGER

Settings in this group govern the behavior of generic actions in the debugger. These controls are then used in conjunction with other processor-specific controls.

See also:

- *Command*
- *Disassembler* on page A-3
- *Memory_maps* on page A-4
- *Board_file* on page A-5.

A.1.1 Command

Settings in this group control the behavior and appearance of the Output view. Use these to:

- customize the input and output format used in this area
- specify the number of lines that are stored in the command history buffer.

When RealView Debugger starts, it uses the last-used settings unless overridden by settings in this group. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect or at next start-up.

Table A-1 describes the settings.

Table A-1 Command settings

Name	Properties
Num_lines	The height of the Output view. The default setting is 5 lines. Values can be entered in hex or decimal, for example 15 or 0x000F.
Radix_in	This setting specifies the format of number input options at start-up. The default format is decimal. However, you can enter hex numbers, for example ce number=0xABCD. Switching to hex also enables you to enter decimal numbers, for example ce number=01234t.
Radix_out	This setting specifies number format output options at start-up. The default format is decimal.
Char	The way that char* and char[] values are displayed, for example as strings or values, for the PRINT command.
Uchar	The way that unsigned char* values are displayed, for example as strings or values, for the PRINT command.
Buffer_height	The number of lines of scrollback available in the Command area. The default is 1024 lines. Values must be greater than 32. Changing this takes effect when RealView Debugger next starts.

A.1.2 Disassembler

Settings in this group control how the disassembly view is displayed in the Code window. This can be set for all processors or for specific processors only. The default settings apply to all processors.

When RealView Debugger starts, it uses the last-used settings unless overridden by these settings. These settings can be overridden dynamically by issuing CLI commands.

Saving changes takes immediate effect.

Note

Some processor disassemblers do not support features configured with these settings, and the settings are ignored.

Table A-2 describes the settings.

Table A-2 Disassembler settings

Name	Properties
Symbols	When instructions reference direct memory locations, either relative to the PC or as absolute references, the debugger tries to show the symbol at that location. Use this to disable this property.
Labels	When an instruction has a label associated with the address, the debugger shows it inline. Use this to disable this property.
Source	By default, high-level source code is interleaved with disassembly code when available. Use this to disable this property.
Asm_source	By default, assembler source code is interleaved with disassembly code when available. This is isolated from high-level language source code display because the assembly source is usually of less interest in this mode. Not all assemblers produce the information to enable assembly tagging. This setting does not affect what is shown in the any source file tabs in the Code window.
Source_line_cnt	By default, interleaved display shows eight lines of source code for any instruction. If there are more source lines associated with the instruction, they are not shown. Use this to define how many lines are shown, or to specify that all are shown.
Stack_syms	Some disassemblers identify frame or stack offset references in the operand fields. Use this to display the corresponding stack-based variable when possible.

Table A-2 Disassembler settings (continued)

Name	Properties
Register_syms	<p>Not available with ARM® tools.</p> <p>Some disassemblers identify register usage in the operand fields. Use this to show the corresponding register-based variable when possible.</p>
Format	<p>Some disassemblers include alternate format which is processor-specific. By default, RealView Debugger shows the format that is most appropriate for the given processor context. Use this to change the default format. Possible values are:</p> <ul style="list-style-type: none"> auto Attempt to auto-detect the disassembly mode. For ARM architecture processors, select from ARM, Thumb®, bytecode, or Thumb-2EE using information from the image file where available. standard Disassemble using the standard instruction format. For ARM architecture processors, this is ARM state. alternate Disassemble using the alternate instruction format. For ARM architecture processors, this is Thumb state. bytecode Disassemble using bytecode instruction format. This is available only for ARM processors. extended Disassemble using Thumb-2EE instruction format. This is available only for ARM processors. <p>You can also change the format dynamically using the DISASSEMBLE command.</p>
Instr_value	In general, disassemblers show instructions as values and as opcodes or operands. Use this to suppress the value display where possible.

See also

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12 for details of the DISASSEMBLE command.

A.1.3 Memory_maps

Contains the setting Access, which indicates the default access type of Default Mapping regions in the **Memory Map** tab of the Process Control view. Possible values are:

- NOM** All Default Mapping regions are defined as No Memory. This is the default. When you specify this option, various RealView Debugger views are affected as follows:
- The Memory view for the Default Mapping regions shows red asterisks. You cannot read from or write to memory locations in these regions using the Memory view.
 - The Disassembly view shows these regions as <Unmapped Memory>.
 - You can load an image into these regions. When you do, a new memory region of type RAM is created. The start address and size of this memory region depends on the load address and the size of the image. The region name is made up of the section names contained in the image, for example: Sect_ER_RO,ER_RW,ER_ZI

- RAM** All Default Mapping regions are defined as RAM.

Note

You must disconnect and reconnect the target for the setting to take effect.

See also

- *How loading an image affects the memory map* on page 9-11

A.1.4 **Board_file**

Change this setting to specify a different board file for the current session:

- If you change this value in the Workspace Options, then it takes immediate effect. The specified board file is read and the contents are used to populate the Connect to Target window.
- If you change this value in the global configuration options, then you must restart RealView Debugger.

Resetting the value back to empty does not take effect until the next time you start RealView Debugger.

Note

Be aware that changing the board file might invalidate any connections listed in the recent connections lists, such as the **Home Page**.

A.2 CODE

The setting in this group controls the display of assembly source code for all Code windows. It does not affect the **Disassembly** tab.

See also:

- *Asm_type*.

A.2.1 Asm_type

When an assembler source file opens, RealView Debugger decides what type of processor is in use. However, the processor type is unknown if there are no active connections.

In this case, RealView Debugger does not know the format of instructions and so cannot define source coloring rules. Therefore, if you open a source file when no connections are established, RealView Debugger displays a selection box where you can specify the processor type.

You can prevent this dialog box being displayed, and force RealView Debugger to use specific source coloring rules. To do this, change *Asm_type* to specify a default processor type on start-up. Set to ARM for ARM architecture-based source code.

Saving a change to this setting takes immediate effect on new assembler source files that you subsequently open.

A.3 ALL

These settings govern the behavior of the editor, the editor display, and access to source code. The settings in the ALL group are used in conjunction with the settings in the DEBUGGER group and the CODE group and might be overridden by settings in either of these two groups.

See also:

- *Text*
- *Search* on page A-10
- *Edit* on page A-11.

A.3.1 Text

Settings in this group control the source code view and editor functions within the Code window.

Saving changes might not take effect until the next time RealView Debugger starts, or when a new Code window opens.

The Text group contains third-level groups and a series of settings:

- | | |
|---------------|---|
| Height | Use this setting to specify the height, in number of lines, for text displayed in the source code view. |
| Width | Use this setting to specify the width, in number of characters, for text displayed in the source code view. |

Src_color_dis

Source coloring is used to make it easier to read source of high-level and low-level languages. All source coloring can be disabled in which case all text is the same color (usually black).

By default, source coloring is enabled, that is this setting is *False*.

Internationalization

Settings in this group configure multiple language support.

Table A-3 describes these settings.

Table A-3 Internationalization settings

Name	Properties
Enabled	Use this to enable or disable internationalization. By default, internationalization is disabled, that is this setting is <i>False</i> .
Language	Use this to specify the language to use for text. The options are: <ul style="list-style-type: none"> • English (default) • Japanese.
Default_encoding	Use this to specify the default text encoding. The options are: <ul style="list-style-type: none"> • ASCII (default) • UTF-8 • Locale.

See also:

- the following in the *RealView Debugger Essentials Guide*:
 - *Localizing the RealView Debugger interface* on page 2-19.

Font_information

This group contains the `Pane_font` setting to set the font used in views:

- On Windows, a Font dialog box is displayed to enable you to change the font settings. shows the default font settings.

Table A-4 Default font settings

Setting	Default Value
Font	Courier New
Font style	Regular
Size	9
Script	Western

- On Red Hat Linux, changing fonts with the `Pane_font` setting is not supported. Table A-4 shows the default font settings that are used.

Source_coloring

These settings control the colors used to identify source tokens. The defaults have been chosen to be easy to read and work well to isolate different program areas. The coloring choices are made relative to the built-in color models.

If you attempt to set the same foreground and background color, then RealView Debugger uses the foreground color but uses the default color for the background.

Table A-5 describes the settings.

Table A-5 Source_coloring settings

Name	Properties
File_extensions	The standard C/C++ source coloring is auto-enabled based on file extension. Use this to specify a comma-separated list of file extensions that, when loaded, trigger source code coloring.
Scheme	The color scheme you want to use for source coloring. You can select one of: <ul style="list-style-type: none"> • Default for the default coloring scheme • VisualStudio for the Visual Studio coloring scheme • CodeWarrior for the CodeWarrior coloring scheme • RVD.1.7 for the RealView Debugger v1.7 coloring scheme.
Numbers_text	Use this to specify the foreground color for numbers displayed in the source code view.
Numbers_bkgrnd	Use this to specify the background color for numbers displayed in the source code view.
Strings_text	Use this to specify the foreground color for strings displayed in the source code view.
Strings_bkgrnd	Use this to specify the background color for strings displayed in the source code view.

Table A-5 Source_coloring settings (continued)

Name	Properties
Keywords_text	Use this to specify the foreground color for C/C++ keywords displayed in the source code view.
Keywords_bkgrnd	Use this to specify the background color for C/C++ keywords displayed in the source code view.
Comments_text	Use this to specify the foreground color for comments displayed in the source code view.
Comments_bkgrnd	Use this to specify the background color for comments displayed in the source code view.
Identifiers_text	Use this to specify the foreground color for identifiers displayed in the source code view.
Identifiers_bkgrnd	Use this to specify the background color for identifiers displayed in the source code view.
User_text	Use this to specify the foreground color for user-defined keywords displayed in the source code view.
User_bkgrnd	Use this to specify the background color for user-defined keywords displayed in the source code view.
Preprocessor_text	Use this to specify the foreground color for preprocessor keywords (#keyword) displayed in the source code view.
Preprocessor_bkgrnd	Use this to specify the background color for preprocessor keywords (#keyword) displayed in the source code view.
Operator_text	Use this to specify the foreground color for operators displayed in the source code view.
Operator_bkgrnd	Use this to specify the background color for operators displayed in the source code view.
User_keywords	Use this to specify a list of user-defined keywords that are highlighted when they appear in the source code view.

A.3.2 Search

Settings in this group control the searching behavior when working with source files in the Code window.

These settings can be overridden dynamically using the menus and toggles in the source code view.

Table A-6 describes these settings.

Table A-6 Search settings

Name	Properties
Direction	Use this to specify the search direction. The default is to search forwards, that is, from the top to the bottom of the file.
Wrap	Use this to specify search behavior when the end of file is reached. The default is to wrap during a search, that is, to search to the end of the file and then to start again at the top until the starting point is reached.
Sensitive	Use this to specify whether uppercase and lowercase characters are treated as identical in searches. By default, searches are case-sensitive.
Regexp	When set to True, full grep-style regular expressions are used in searches. The default is False, not enabled.
Fail	Use this to specify editor behavior when a search fails. Set to dialog by default, you can change this to flash.

A.3.3 Edit

Settings in this group control general editor behavior when working with source files in the Code window.

These settings can be overridden dynamically using the menus and toggles in the source code view.

The Edit group contains three third-level groups and a series of settings:

Backup

These settings control the backup behavior when working with source files in the Code window. Table A-7 describes these settings.

Table A-7 Backup settings

Name	Properties
Disable	By default, a backup file is created when a file is edited. This provides a useful safety feature. Use this to disable this feature if required.
Backup_dir	By default, backup files are saved in the same directory as the original file. Use this to specify a path name to a new location, for example to keep all backup files in one special directory.
	<p>Note</p> <p>If a backup directory is specified then the files are saved there using the original file extension.</p>
Backup_ext	By default, backup files are saved with the .bak extension appended to the original filename.
	<p>Note</p> <p>The file extension is only used if a backup directory is not specified.</p>

Tab_conv

Settings in this group control the display behavior when working with source files in the Code window. These settings are used to handle tabs and spaces.

Tabs are permitted in files and are left untouched, by default. Use these settings to convert tabs to spaces when writing to the file, that is saving, and to convert spaces to tabs when reading the file.

Spaces are not converted to tabs inside " and " quoting blocks on a line. Table A-8 describes these settings.

Table A-8 Tab_conv settings

Name	Properties
Tabs_to_spaces	Converts tabs to spaces when the file is saved.
Spaces_to_tabs	Converts spaces to tabs when the file is read.
To_spaces_ext	Use this to specify file extensions where tab conversions take place. Specify a list separated by semi-colons (:).
To_tabs_ext	Use this to specify file extensions where space conversions take place. Specify a list separated by semi-colons (:).

Src_ctrl

This group is deprecated.

Edit

Settings in this group configure editor behavior when working with source files in the Code window.

Table A-9 describes these settings.

Table A-9 Edit settings

Name	Properties
Drag_drop_dis	Use this to disable drag-and-drop editing when working in a source code view.
Vi	Deprecated.
Indent	Use this to set indenting so that a specified number of spaces are inserted as you open a new line. By default, auto-indent inserts the same number of spaces as on the previous line. If the previous line is a left curly bracket ({) the shift is increased. If the previous line is a right curly bracket (}), shift spaces are subtracted.
Undo	Use this to specify the levels of undo and redo. By default, this is set to 64.
Tab	Use this to specify the size of TAB settings when working in the File Editor. By default, this is set to 8. Use a value between 1 and 16.
Shift	Use this to specify the size of shift spaces as used in the Indent rule and accessed through the Code window Edit menu options. By default, this is set to 2. Use a value between 2 and 32.
Line_number	By default, line numbering is enabled in the debugger and the File Editor. Use this to change the editor default to hide line numbers at start-up.
No_tooltip	By default, tooltip evaluation of variables and registers is enabled. Change this setting to True to disable this feature.
Timer	During file editing, the editor periodically checks to see if another tool has edited or deleted the files being tested. A warning is shown if an update is detected. Use this to specify the number of seconds between checks. The default is 60 seconds. Use values greater than 30 seconds. Set to -1 to disable this feature.
Tool_save	When performing a build, you are prompted to resave any files that have been edited. Use this to specify automatic resaving of changed files at build time to ensure that your latest sources are included. You can also set a no-save, no-ask value.

Table A-9 Edit settings (continued)

Name	Properties
Startup	The default start-up file, that is rvdebug.sav in your home directory, contains a list of previously edited files and information from previous debugging or editing sessions. This enables historical information to be separated from your current session. Use this to specify a different start-up file, in a new location. Set this to - (dash) to specify that no start-up file is used.
Template	During file editing, you can use templates to speed up code development. The template file contains templates that you can use or edit as required. By default, the file is named rvdebug.tpl and is saved in your home directory, or in your default settings directory. Use this to change this path name.
Restore_state	Not used.

A.4 CONNECTION

Available only in the workspace settings.

When you exit RealView Debugger, you have the option to save connection details so that the same connections are used when RealView Debugger starts up with the same workspace.

This is a special group, to specify connections, maintained by RealView Debugger. An entry is created for each connection. Entries cannot be edited.

— Note —

You must not delete these entries.

See also:

- *Viewing the current workspace settings* on page 17-11.

A.5 WINDOW

Available only in the workspace settings.

This is a special group of windows internals maintained by RealView Debugger. An entry is created for each open window. Entries cannot be edited.

— Note —

You must not delete these entries.

See also:

- *Viewing the current workspace settings* on page 17-11.

Appendix B

Configuration Files Reference

This appendix describes the files set up for a new installation of RealView® Debugger, 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 B-2
- *Files in the default settings directory* on page B-3
- *Files in the home directory* on page B-5.

B.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.

When you run the debugger for the first time, some of this information is copied into the RealView Debugger default home directory.

— Note —

This appendix describes configuration files created (or modified) when you install RealView Debugger on Windows.

See also:

- Appendix E *RealView Debugger on Red Hat Linux*.

B.2 Files in the default settings directory

When you first run RealView Debugger after installation, files containing default settings and target configuration details are copied to your default settings directory:

```
C:\Documents and Settings\userID\Local Settings\Application  
Data\ARM\rvdebug\version\shadowbase\etc
```

The files in the default settings directory are:

armul.var	A list of <i>RealView ARMulator® ISS</i> (RVISS) variants used by RealView Debugger. You can edit this file to add your own variants of the RVISS models.
pARM*.prc	Processor-specific information files used to define support for emulators and simulators, for example pARM.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 any Debug Interface configuration files, such as *.rvc.
rvdebug.tpl	Template file used to contain standard templates. You can edit these files for use in source files.
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	Send a Problem Report (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, rename them to distinguish the copies from the originals, and edit your copies using the Connection Properties window, for example CM940T.bcd. Do not edit these files manually.
*.cm1	Configuration files for Model Library connections. You change the contents of these files when you modify the configuration of a Model Library connection using the Model Configuration Utility dialog box. Do not edit these files manually.
*.cmp	Configuration files for Model Process connections. You change the contents of these files when you modify the configuration of a Model Process connection using the Model Configuration Utility dialog box. Do not edit these files manually.
*.rvc	Configuration files for DSTREAM or RealView ICE connections. You change the contents of one of these files when you modify the configuration of a DSTREAM or RealView ICE connection using the RVConfig utility. Do not edit these files manually.
*.smc	Configuration files for <i>Instruction Set System Model</i> (ISSM), <i>Real-Time System Model</i> (RTSM) and SoC Designer connections. You change the contents of these files when you modify the configuration of an ISSM or RTSM connection using the Model Configuration Utility dialog box.

Do not edit these files manually.

`*.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 define options such as workspace settings, for example `aws.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.

See also:

- *Files in the home directory* on page B-5
- *DSTREAM and RealView ICE Using the Debug Hardware Configuration Utilities*
- *RealView ARMulator ISS User Guide*.

B.3 Files in the home directory

When you run RealView Debugger for the first time, files containing default settings and target configuration details are copied into your RealView Debugger default home directory where you can modify them:

C:\Documents and Settings\username\Application Data\ARM\rvdebug\version

The files in this directory are:

- armreg.sig** An internal settings file that is created or updated each time you run RealView Debugger.
Do not edit this file manually.
- exphist.sav** 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.
- rvdebug.aws** 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.
- rvdebug.brd** Your board file containing target configuration settings. For a new installation, this is a duplicate of the file installed in the default settings directory.
This file references .rbe files and .bcd files.
- rvdebug.ini** 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.
- rvdebug.sav** 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.
- settings.sav** This file is used to persist various settings when between RealView Debugger sessions.
- *.auc** These are RVISS target configuration settings files created when you first run RealView Debugger, for example default.auc.
- *.bcd** Board/Chip definition files that you have copied into your home directory from the default settings directory. It is suggested that you do this when you want to create your own versions based on the installed BCD files.
- *.smc** These are ISSM and RTSM target configuration settings files created when you first run RealView Debugger, for example ISSM_0.smc.

See also:

- *Backup files*
- *Files in the default settings directory* on page B-3.

B.3.1 Backup files

When you change a configuration file in your home directory, RealView Debugger makes a backup copy of the current version to enable you to restore your previous settings. By default, backup files are given the .bak extension and are stored in the same location as the original file, for example rvdebug.aws.bak and rvdebug.brd.bak. You can change this behavior in your workspace.

See also:

- *Files in the default settings directory* on page B-3
- *Files in the home directory* on page B-5.

Appendix C

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 C-2
- *RealView Debugger operations* on page C-5
- *Comparison of RealView Debugger and AXD commands* on page C-7
- *Converting legacy AXD scripts to RealView Debugger format* on page C-11.

C.1 RealView Debugger configuration

The following sections answer FAQs about configuration options in RealView Debugger:

- *Where do I find Debugger Internal variables?*
- *How do I set top_of_memory?*
- *How do I configure vector_catch? on page C-3*
- *How do I configure semihosting? on page C-4.*

C.1.1 Where do I find Debugger Internal variables?

Select **Registers** from the **View** menu to display the Registers view. The RealView Debugger internals are available from:

- the **CycleCount** and **Semihost** tabs for connections to *RealView ARMulator® ISS* (RVISS)
- the **Debug** tab for all other connections.

Equivalent symbols for RVISS targets

Table C-1 lists the equivalents of some important symbols for RVISS targets.

Table C-1 Equivalents of some important symbols for RVISS targets

AXD Symbol	RealView Debugger symbol
\$memstats	There is no equivalent to the \$memstats array. However, the individual memory map related values are available as @mapfile_symbolname symbols. To see a list of the symbols, enter the following CLI command: REGINFO,access,match:mapfile
\$rdi_log	@RVISS_log
\$statistics	There is no equivalent to the \$statistics structure. However, the individual statistics related values are available as @stats_symbolname symbols. To see a list of the symbols, enter the following CLI command: REGINFO,access,match:stats

C.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.

— Note —

The `top_of_memory` symbol is not supported on RVISS, ISSM, and SoC Designer targets.

If `top_of_memory` is not set, RealView Debugger sets it to a default value of `0x20000`. A warning is displayed in the **Cmd** and **Log** tabs:

Warning: No stack/heap or top_of_memory defined - setting top_of_memory to 0x00020000

The value of `top_of_memory` can be overridden for a single debug session from the **Debug** tab or **CycleCount** tab in the Registers view.

You can also set the value of `top_of_memory` for a particular target connection using the Connection Properties.

Note

To use the new setting, you must 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.

See also

- *Configuring top of memory* on page 3-23
- *Where do I find Debugger Internal variables?* on page C-2
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Setting top of memory* on page 4-26.

C.1.3 How do I configure vectorCatch?

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 `vectorCatch` value.

The value of `vectorCatch` represents a bit field, where a set bit corresponds to a trapped exception. Bits 0 to 8 corresponds to the vectors Reset to Error, respectively. Although the value can only be displayed in hexadecimal, you can enter values in binary format using the notation `0b`. The default value of `0x13B` (`0b100111011`) corresponds to trapping as shown in Table C-2.

Table C-2 Trapped Processor Exceptions defaults

Exception	Trapped	Comment
Reset	Yes	-
Undefined	Yes	-
SVC	No	<i>SuperVisor Call</i> (SVC) vector can also be trapped by the debugger to enable standard semihosting
Prefetch Abort	Yes	-
Data Abort	Yes	-
Reserved (Address)	Yes	Catch Address exceptions. Used only by the obsolete 26-bit ARM processor architectures.
IRQ	No	-
FIQ	No	-
Error	Yes	Catch Error exceptions. Supported only for RVISS targets.

You can also set the state of individual vector catches for a particular target connection using the Connection Properties.

Note

To use the new setting, you must connect to the target.

If a Board/Chip definition file is selected for this connection, then this file might contain a value for `vectorCatch` that overrides the target connection setting.

See also

- *Configuring vector catch* on page 3-22
- *Where do I find Debugger Internal variables?* on page C-2.
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring vector catch* on page 3-24.

C.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, in the **StdIO** tab of the Output view. You can enable or disable semihosting from the debugger.

You can set semihosting for a single debug session from the **Debug** or **Semihost** tab of the Registers view, depending on your target. The available options, if you are using DSTREAM or RealView ICE, are NO and STD (Standard).

You can also set semihosting for a particular target connection using the Connection Properties.

Note

To use the new setting, you must 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.

See also

- *Configuring semihosting* on page 3-21
- *Where do I find Debugger Internal variables?* on page C-2.
- the following in the *RealView Debugger Target Configuration Guide*:
 - *Configuring Semihosting* on page 3-29.

C.2 RealView Debugger operations

The following sections answer FAQs about debugging with RealView Debugger:

- *Viewing Coprocessor variables*
- *How do I load symbols for an image?*
- *RealView ARMulator ISS Benchmarking*
- *Debug illusion issues.*

C.2.1 Viewing Coprocessor variables

The RealView Debugger Registers view includes an additional tab giving you visibility of coprocessor registers such as CP15. Consult the appropriate *Technical Reference Manual* (TRM) for your ARM processor to obtain information on the function of bit fields within each CP15 register.

C.2.2 How do I load symbols for an image?

To load only the symbols for an image in RealView Debugger:

1. Select **Load Image...** from the **Target** menu to display the Load File to Target dialog box.
2. Make sure that the Symbols Only check box is selected.

See also

- *Loading symbols only for an image* on page 4-16.

C.2.3 RealView ARMulator ISS Benchmarking

Currently RealView Debugger does not support RVISS benchmarking, as described in *ARM Application Note 93*.

C.2.4 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`.

C.3 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.

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 C-3 for the equivalent RealView Debugger command.

Table C-3 AXD commands and equivalents

AXD commands	RealView Debugger commands
BACKTRACE	<u>WHERE</u>
<u>BREAK</u>	<u>BREAKEXECUTION</u> <u>BREAKINSTRUCTION</u> <u>DTBREAK</u>
<u>CCLASSES</u>	<u>BROWSE</u>
<u>CFUNCTIONS</u>	<u>BROWSE</u>
<u>CLASSES</u>	<u>BROWSE</u>
CLEAR	<u>VCLEAR</u>
CLEARSTAT	-
CLEARWATCH	<u>CLEARBREAK</u>
<u>COMMENT string</u>	<u>EPRINTF</u>
<u>CONTEXT</u>	<u>CONTEXT</u> <u>WHERE</u>
<u>CONVARIABLES</u>	<u>EXPAND</u> <u>SCOPE</u>
<u>CVARIABLES</u>	<u>BROWSE</u>
DBGINTERNAL	<u>EDITBO</u>
<u>DISASSEMBLE</u>	<u>DISASSEMBLE</u> <u>MODE</u>
ECHO	-
EXAMINE	<u>DUMP</u>
<u>FILES</u>	-
FILLMEM	<u>FILL</u> <u>SETMEM</u>
FINDSTRING	<u>SEARCH</u>
FINDVALUE	<u>SEARCH</u>
FORMAT	<u>OPTION RADIX</u> <u>SETTINGS DISASM</u> <u>SETTINGS DSMVALUE</u>

Table C-3 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
<u>FUNCTIONS</u>	<u>PRINTSYMBOLS</u>
<u>GETFILE</u>	<u>READFILE</u>
<u>GO</u>	<u>GO</u>
<u>HELP</u>	<u>DCOMMANDS</u> <u>HELP</u>
<u>IMAGES</u>	<u>DTFILE</u>
<u>IMGPROPERTIES</u>	<u>DTFILE</u>
<u>IMPORTFORMAT</u>	-
<u>IN</u>	<u>UP</u>
<u>LET</u>	<u>CEXPRESSION</u> <u>SETMEM</u>
<u>LIST</u>	<u>DISASSEMBLE</u> <u>MODE</u> <u>DUMP</u>
<u>LISTFORMAT</u>	-
<u>LOAD</u>	<u>LOAD</u> <u>RELOAD</u>
<u>LOADBINARY</u>	<u>READFILE</u>
<u>LOADSESSION</u>	<u>READBOARDFILE</u>
<u>LOADSYMBOLS</u>	<u>LOAD/NS</u>
<u>LOG</u>	<u>JOURNAL</u>
<u>LOWLEVEL</u>	<u>PRINTSYMBOLS</u>
<u>MEMORY</u>	<u>DUMP</u>
<u>OBEY</u>	<u>INCLUDE</u>
<u>OUT</u>	<u>DOWN</u>
<u>PARSE</u>	-
<u>PRINT</u>	<u>PRINTF</u> <u>CEXPRESSION</u> <u>PRINTVALUE</u>
<u>PROCESSORS</u>	<u>DTPROCESS</u>
<u>PROCPROPERTIES</u>	-
<u>PUTFILE</u>	<u>WRITEFILE</u>
<u>QUITDEBUGGER</u>	<u>QUIT</u>
<u>READSYMS</u>	<u>LOAD/NS</u>
<u>RECORD</u>	<u>LOG</u>

Table C-3 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
REGBANKS	-
REGISTERS	-
RELOAD	<u>RELOAD</u>
RUN	<u>GO</u>
RUNTOPOS	<u>GO</u>
SAVEBINARY	<u>WRITEFILE</u>
SAVESESSION	-
SETACI	-
SETBREAKPROPS	<u>BREAKEXECUTION</u> <u>BREAKINSTRUCTION</u>
SETIMGPROP "file" cmdline	<u>ARGUMENTS</u>
SETMEM	<u>CEXPRESSION</u> <u>FILL</u> <u>SETMEM</u>
SETPC	<u>SETREG @PC</u>
SETPROC	BOARD THREAD
SETPROCPROP	-
SETPROCPROP vector_catch	BGLOBAL
SETPROCPROP semihosting_enabled	
SETREG	<u>SETREG</u>
SETSOURCEDIR	-
SETWATCH	<u>CEXPRESSION</u> <u>SETMEM</u>
SETWATCHPROPS	<u>BREAKEXECUTION</u> <u>BREAKINSTRUCTION</u>
SOURCE	<u>LIST</u>
SOURCEDIR	-
STACKENTRIES	<u>WHERE</u>
STACKIN	UP
STACKOUT	<u>DOWN</u>
STATISTICS	STATS
STEP	<u>STEPINSTR</u> <u>STEPLINE</u> <u>STEPINSTR</u> STEP0

Table C-3 AXD commands and equivalents (continued)

AXD commands	RealView Debugger commands
STEPSIZE	RUN
STOP	STOP
TYPE	LIST
UNBREAK	CLEARBREAK
UNWATCH	CLEARBREAK
VARIABLES	PRINTTYPE
WATCHPT	BREAKACCESS BREAKREAD BREAKWRITE
WHERE	CONTEXT WHERE

See also:

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

C.4 Converting legacy AXD scripts to RealView Debugger format

The following section describes how you can convert legacy AXD scripts to the RealView Debugger format:

- *axd2rvd command syntax.*

The *RealView Development Suite* (RVDS) product includes the program `axd2rvd` to help with this process.

C.4.1 `axd2rvd` command syntax

The commands have the syntax:

```
axd2rvd [-Vsn] [-A] -I infile.asd [-O rvdfile.txt]
```

where:

-I <i>infile.asd</i>	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 <i>rvdfile.txt</i>	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 this program.

Note

These commands help you convert a script file to a RealView Debugger include file, but because of the different features provided by AXD, and the very different command set implemented, you must check the resulting file for commands that the script cannot convert.

See also

- *Comparison of RealView Debugger and AXD commands* on page C-7.

Appendix D

Moving from armsd to RealView Debugger

This appendix is aimed at users *ARM® Symbolic Debugger* (armsd) to RealView® Debugger. It contains the following sections:

- *RealView Debugger configuration* on page D-2
- *Comparison of RealView Debugger and armsd commands* on page D-3
- *Converting legacy armsd scripts to RealView Debugger format* on page D-6.

D.1 RealView Debugger configuration

The following section answers FAQs about configuration options in RealView Debugger:

- *Where do I find Debugger Internal variables?*

D.1.1 Where do I find Debugger Internal variables?

Select **Registers** from the **View** menu to display the Registers view. The RealView Debugger internals are available from:

- the **CycleCount** and **Semihost** tabs for connections to *RealView ARMulator® ISS* (RVISS)
- the **Debug** tab for all other connections.

Equivalent symbols for RVISS targets

Table D-1 lists the equivalents of some important symbols for RVISS targets.

Table D-1 Equivalents of some important symbols for RVISS targets

armsd Symbol	RealView Debugger symbol
\$memstats	There is no equivalent to the \$memstats array. However, the individual memory map related values are available as @mapfile_symbolname symbols. To see a list of the symbols, enter the following CLI command: REGINFO,access,match:mapfile
\$rdi_log	@RVISS_log
\$semihosting_enabled	@semihost_state For a list of other semihosting related symbols, enter the following CLI command: REGINFO,access,match:semihost
\$statistics	There is no equivalent to the \$statistics structure. However, the individual statistics related values are available as @stats_symbolname symbols. To see a list of the symbols, enter the following CLI command: REGINFO,access,match:stats
\$statistics_inc	-

D.2 Comparison of RealView Debugger and armsd commands

Look up any armsd command in Table D-2 for the equivalent RealView Debugger command. For full details of the commands available in armsd, see your *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 D-2 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	<i>function_name(params)</i> as an expression
COMMENT	<u>EPRINTF</u>
CONTEXT	<u>SCOPE</u>
Control-C	STOP
COPROC	-
CREGDEF	Edit board file
CREGISTERS	@ <i>regname</i> as an expression.
CWRITE	<u>SETREG</u>
EXAMINE	<u>DUMP</u>
FIND	<u>SEARCH</u>
FPREGISTERS	@ <i>regname</i> for each register as an expression
GETFILE	<u>READFILE</u>
GO	<u>GO</u>
HELP	<u>DCOMMANDS</u> HELP
IN	<u>DOWN</u>

Table D-2 armsd commands and equivalents (continued)

armsd commands	RealView Debugger commands
ISTEP	<u>STEPINSTR</u> <u>STEPOINSTR</u>
LANGUAGE	-
LET	<u>SETREG</u> <u>CEXPRESSION</u>
LIST	<u>DISASSEMBLE</u> <u>PRINTDSM</u> <u>MODE</u> <u>DUMP</u>
LOAD	<u>LOAD</u>
LOG	<u>LOG</u>
LSYM	<u>PRINTSYMBOLS</u>
<i>1value_memoryaddress = data_value</i>	<u>CEXPRESSION</u>
OBEY	<u>INCLUDE</u>
OUT	<u>UP</u>
PAUSE	<u>PAUSE</u>
PRINT	<u>CEXPRESSION</u> <u>PRINTF</u> <u>PRINTVALUE</u>
PROFCLEAR	-
PROFOFF	-
PROFON	-
PROFWRITE	-
PUTFILE	<u>WRITEFILE</u>
QUIT	<u>QUIT</u>
READSYMS	<u>LOAD/NS</u>
REGISTERS	@ <i>regname</i> for each register as an expression
RELOAD	<u>RELOAD</u>
RETURN	-
STEP	<u>STEPLINE</u> <u>STEP0</u>
SYMBOLS	<u>EXPAND</u> <u>PRINTSYMBOLS</u>
TYPE	<u>LIST</u>
UNBREAK	<u>CLEARBREAK</u>

Table D-2 armsd commands and equivalents (continued)

armsd commands	RealView Debugger commands
UNWATCH	<u>CLEARBREAK</u>
VARIABLE	<u>PRINTTYPE</u>
WATCH	<u>BREAKACCESS</u> <u>BREAKREAD</u> <u>BREAKWRITE</u>
WHERE	<u>CONTEXT</u>
WHILE	<u>WHILE</u> statement (in macro or include file).

See also:

- the following in the *RealView Debugger Command Line Reference Guide*:
 - *Alphabetical command reference* on page 2-12.

D.3 Converting legacy armsd scripts to RealView Debugger format

The following section describes how you can convert legacy armsd scripts to the RealView Debugger format:

- *armsd2rvd command syntax.*

The *RealView Development Suite* (RVDS) product includes the program `armsd2rvd` to help with this process.

D.3.1 armsd2rvd command syntax

The command has the syntax:

```
armsd2rvd [-Vsn] [-A] -I infile.asd [-O rvdfile.txt]
```

where:

<code>-I infile.asd</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 rvdfile.txt</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 these programs.

Note

This command helps you convert a script file to a RealView Debugger include file, but because of the different features 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 also

- *Comparison of RealView Debugger and armsd commands* on page D-3.

Appendix E

RealView Debugger on Red Hat Linux

This appendix provides supplementary information for developers using RealView® Debugger on Red Hat Linux. It contains the following sections:

- *About this Appendix* on page E-2
- *Getting more information* on page E-3
- *Changes to target configuration details* on page E-4
- *Changes to GUI and general user information* on page E-5.

E.1 About this Appendix

This appendix describes features that are specific to using RealView Debugger v3.1 on Red Hat Linux, and contains corrections and additions to the documentation suite.

E.2 Getting more information

The following sections describe how to get more information on RealView Debugger:

- *Online resources*
- *Feedback on RealView Debugger.*

E.2.1 Online resources

ARM® Limited provides a range of services to support developers using RealView Debugger. Among the downloads available are:

- enhanced support for different hardware platforms through technical information and board description files
- product Release Notes
- updates to documentation
- Frequently Asked Questions.

You can access these resources from the ARM web site at <http://www.arm.com>.

E.2.2 Feedback on RealView Debugger

If you have any problems with RealView Debugger, submit a *Software Problem Report* (SPR):

1. Select **Send a Problem Report...** from the RealView Debugger **Help** 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.

E.3 Changes to target configuration details

The following sections describe changes to the target configuration features:

- *Default configuration files*
- *Target configuration entries.*

E.3.1 Default configuration files

Default configuration files are available in:

install_directory/RVD/Core/.../linux-pentium/etc

These files are:

- board file, for example rvdebug.brd
- Debug Interface configuration files, such as rvi.rvc
- Board/Chip definition files, for example CM940T.bcd.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - Chapter 3 *Customizing a Debug Configuration*
 - Chapter 4 *Configuring Custom Memory Maps, Registers and Peripherals*.

E.3.2 Target configuration entries

RVISS enables you to connect to ARM7, ARM9, and ARM11 simulated targets.

See also

- the following in the *RealView Debugger Target Configuration Guide*:
 - *About customizing a DSTREAM or RealView ICE Debug Interface configuration* on page 2-3.

E.4 Changes to GUI and general user information

The following sections describe changes to the GUI and general user information:

- *Column resizing*
- *Saving favorites* on page E-6.

E.4.1 Column resizing

When using the Code window, the following views use two columns to display debug data:

- Break/Tracepoints
- Call Stack
- Locals
- Process Control
- Watch.

Note

You cannot change the size of columns in the Resource Viewer.

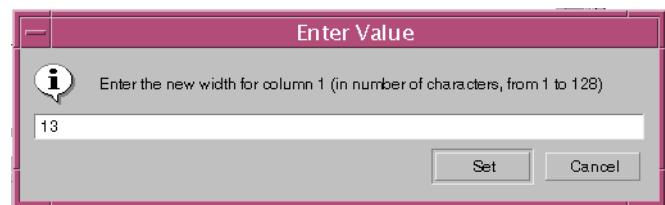
To make the display easier to read, you can change the size of the first column using a new menu option available only on Red Hat Linux:

1. Select **Reload Image to Target** from the **Target** menu to reload the image `dhrystone.axf`.
2. Click the **Locate PC** button on the Debug toolbar to view the source file `dhry_1.c`.
3. Set a simple breakpoint by double-clicking on line 183.
4. Click **Run** to start execution.
5. Enter `5000` when asked for the number of runs.
6. If you do not have a Watch view visible, select **Watches** from the **View** menu to display the Watch view.
7. Right-click somewhere in the header columns to display the context menu, shown in Figure E-1.



Figure E-1 Column resizing in the Watch view

8. Select the option **Set Width Of Column 1** to display the prompt box where you can specify the size you want, shown in Figure E-2 on page E-6.

**Figure E-2 Column resizing prompt box**

When the prompt box first appears, it contains the current setting. Enter a value between 1 and 128.

9. Either:
 - Click **Set** to confirm the new setting.
 - Click **Cancel** to leave the size unchanged.

Note

You can use the normal column resizing controls on the Symbols view and the Connect to Target window.

If you resize columns be aware of the following:

- The new column size holds only for the current window, and only in the current session. Default sizes are restored at each start up.
- You can only resize the first column in any two-column display.
- Columns can only be adjusted to within one monospaced character position.
- The second column is automatically set to accommodate the longest item.
- Changing the column size applies to all tabs in a multitab display, that is the Watch, Locals, and Process Control views.

E.4.2 Saving favorites

Be aware of the following when working on Red Hat Linux:

- The history file, `exphist.sav`, is only created if you create and save a favorite, for example a breakpoint or watchpoint.
- The history file is not controlled in the same way when you access an **Open...** dialog box.

When the file has been created and saved, it is updated at the end of each debugging session in the usual way.