

RealView® Developer Kit

Version 2.2

Debugger User Guide



RealView Developer Kit

Debugger User Guide

Copyright © 2005, 2006 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
April 2005	B	Release for RVDK v2.2
January 2006	C	Includes trace support

Proprietary Notice

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

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

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

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

Confidentiality Status

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

Product Status

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

Web Address

<http://www.arm.com>

Contents

RealView Developer Kit Debugger User Guide

	Preface	
	About this book	viii
	Feedback	xiii
Chapter 1	Starting to use RealView Debugger	
1.1	Starting RealView Debugger	1-2
1.2	RealView Debugger directories	1-6
Chapter 2	Connecting to Targets	
2.1	Working with the Connection Control window	2-2
2.2	Managing connections	2-6
2.3	Connecting to a target	2-10
2.4	Failing to make a connection	2-16
2.5	Disconnecting from a target	2-18
Chapter 3	Working with Images	
3.1	Loading images	3-2
3.2	Managing images	3-9
3.3	Working with symbols	3-18
3.4	Working with multiple images	3-19
3.5	Unloading and reloading images	3-21

Chapter 4	Controlling Execution	
4.1	Submitting commands	4-2
4.2	Defining execution context	4-3
4.3	Using execution controls	4-7
4.4	Working with the Debug menu	4-10
4.5	Automating debugging operations	4-13
4.6	Searching for source files	4-17
Chapter 5	Working with Breakpoints	
5.1	Breakpoints in RealView Debugger	5-2
5.2	Setting default breakpoints	5-14
5.3	Generic breakpoint operations	5-19
5.4	Setting unconditional breakpoints explicitly	5-20
5.5	Setting hardware breakpoints explicitly	5-22
5.6	Specifying processor exceptions (global breakpoints)	5-31
5.7	Setting conditional breakpoints	5-32
5.8	Using the Set Address/Data Breakpoint dialog box	5-39
5.9	Attaching macros to breakpoints	5-52
5.10	Controlling the behavior of breakpoints	5-55
5.11	Using the Break/Tracepoints pane	5-59
5.12	Disabling and clearing breakpoints	5-67
5.13	Setting breakpoints from saved lists	5-69
Chapter 6	Memory Mapping	
6.1	About memory mapping	6-2
6.2	Enabling and disabling memory mapping	6-4
6.3	Setting up a memory map	6-5
6.4	Viewing the memory map	6-7
6.5	Editing map entries	6-12
6.6	Setting top_of_memory and stack values	6-13
6.7	Generating linker command files for non-ARM targets	6-14
Chapter 7	Working with Debug Views	
7.1	Working with registers	7-2
7.2	Working with memory	7-11
7.3	Working with the stack	7-22
7.4	Using the call stack	7-28
7.5	Working with watches	7-32
Chapter 8	Reading and Writing Memory, Registers and Flash	
8.1	About interactive operations	8-2
8.2	Using the Memory/Register Operations menu	8-3
8.3	Accessing interactive operations in other ways	8-4
8.4	Working with Flash	8-5
8.5	Examples of interactive operations	8-10

Chapter 9	Working with Browsers	
9.1	Using browsers	9-2
9.2	Using the Data Navigator pane	9-5
9.3	Using the Symbol Browser pane	9-17
9.4	Using the Function List dialog box	9-19
9.5	Using the Variable List dialog box	9-21
9.6	Using the Module/File List dialog box	9-23
9.7	Using the Register List Selection dialog box	9-25
9.8	Specifying browser lists	9-27
9.9	Using the Favorites Chooser/Editor dialog box	9-29
 Chapter 10	 Working with Macros	
10.1	About macros	10-2
10.2	Using macros	10-8
10.3	Getting more information	10-17
 Chapter 11	 Configuring Workspace Settings	
11.1	Using workspaces	11-2
11.2	Viewing workspace settings	11-8
11.3	Configuring workspace settings	11-14
 Chapter 12	 Connection and Target Configuration	
12.1	About target connections and configuration	12-2
12.2	Viewing board file properties	12-5
12.3	Configuration files	12-11
 Chapter 13	 Configuring Custom Targets	
13.1	About target configuration	13-2
13.2	Linking a board, chip, or component to a connection	13-6
13.3	Creating new target descriptions	13-14
13.4	Example descriptions	13-22
 Chapter 14	 Configuring Custom Connections	
14.1	Working with connection properties	14-2
14.2	Working with RVI/RVI-ME targets	14-7
 Chapter 15	 RTOS Support	
15.1	About Real Time Operating Systems	15-2
15.2	Using RealView Debugger RTOS extensions	15-7
15.3	Connecting to the target and loading an image	15-22
15.4	Associating threads with views	15-27
15.5	Working with OS-aware images in the Process Control pane	15-34
15.6	Using the Resource Viewer window	15-41
15.7	Debugging your RTOS application	15-47
15.8	Using CLI commands	15-58

Chapter 16	Programming Flash with RealView Debugger	
16.1	Introduction to Flash programming with RealView Debugger	16-2
16.2	Programming an image into Flash	16-3
16.3	Troubleshooting	16-5
Appendix A	Workspace Settings Reference	
A.1	DEBUGGER	A-2
A.2	CODE	A-6
A.3	ALL	A-8
Appendix B	Configuration Properties Reference	
B.1	About this appendix	B-2
B.2	Target configuration and connections	B-3
B.3	Generic groups and settings	B-5
B.4	Target configuration reference	B-8
B.5	Custom connection reference	B-26
	Glossary	

Preface

This preface introduces the *RealView® Developer Kit v2.2 Debugger User Guide* that shows you how to use RealView Debugger to debug your application programs. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xiii.

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
- a glossary of terms for users new to RealView Debugger.

Intended audience

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

Examples

Your hardware and software might not be the same as those used for testing the examples given in this book, 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 and projects stored in the \Examples directory in your RealView Developer Suite root installation.

Using this book

This book is organized into the following chapters:

Chapter 1 *Starting to use RealView Debugger*

Read this chapter for details of how to start using RealView Debugger on your workstation.

Chapter 2 *Connecting to Targets*

Read this chapter for details on how the Connection Control window is used to view connection details and to configure new ones.

Chapter 3 *Working with Images*

This chapter contains details of how to work with application programs in RealView Debugger, including how to load an image ready for debugging and how to view image details.

Chapter 4 *Controlling Execution*

Read this chapter for details of how to control program execution during your debugging sessions. It gives details on using the major control options and describes how to use files to keep a record of the debugging session.

Chapter 5 *Working with Breakpoints*

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

Chapter 6 *Memory Mapping*

This chapter gives details of how to manage memory for single processor operation during a debugging session. It describes the Process Control pane that contains a dynamic display of the current memory configuration.

Chapter 7 *Working with Debug Views*

Read this chapter for details of how to monitor execution of your application program by setting watches, reading registers and tracking changes to memory contents.

Chapter 8 *Reading and Writing Memory, Registers and Flash*

Read this chapter for details of operations on registers contents and memory that can be accessed dynamically during a debugging session. In this way, RealView Debugger enables you to have greater control over your application software.

Chapter 9 *Working with Browsers*

Read this chapter for details of the browsers accessible from the Code window when using RealView Debugger.

Chapter 10 *Working with Macros*

Read this chapter for details of how to create macros when working with RealView Debugger.

Chapter 11 *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.

Chapter 12 *Connection and Target Configuration*

Read this chapter for details of the connection and target configuration system used by RealView Debugger.

Chapter 13 *Configuring Custom Targets*

Read this chapter for details on how you can describe your debug target to the debugger.

Chapter 14 *Configuring Custom Connections*

Read this chapter for details on how you can configure the connection that RealView Debugger makes to your target.

Chapter 15 *RTOS Support*

Read this chapter for details on the *Real Time Operating System* (RTOS) support that is available in RealView Debugger.

Chapter 16 *Programming Flash with RealView Debugger*

Read this chapter for details on how to use RealView Debugger to program Flash memory on your target hardware.

Appendixes and Glossary

Appendix A *Workspace Settings Reference*

Read this appendix for details of how to set options to configure your working environment using RealView Debugger workspaces. This appendix must be read in association with Chapter 11 *Configuring Workspace Settings*.

Appendix B *Configuration Properties Reference*

This appendix contains reference details about board file entries that define target configurations and custom connections.

Glossary

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

Typographical conventions

The following typographical conventions are used in this book:

<i>italic</i>	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.
monospace	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.

Further reading

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

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

ARM publications

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

- *RealView Developer Kit v2.2 Compiler and Libraries Guide* (ARM DUI 0282)
- *RealView Developer Kit v2.2 Assembler Guide* (ARM DUI 0283)
- *RealView Developer Kit v2.2 Command Line Reference* (ARM DUI 0284)
- *RealView Developer Kit v2.2 Linker and Utilities Guide* (ARM DUI 0285)
- *RealView Developer Kit v2.2 Project Management Guide* (ARM DUI 0291).

For general information on software interfaces and standards supported by ARM, see <http://www.arm.com>.

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

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

- *RealView ICE and RealView Trace User Guide* (ARM DUI 0155)
- *RealView ICE Micro Edition v1.1 User Guide* (ARM DUI 0220).

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 Limited welcomes feedback on both RealView Debugger and its documentation.

Feedback on RealView Debugger

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

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

Feedback on this book

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

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

General suggestions for additions and improvements are welcome.

Chapter 1

Starting to use RealView Debugger

This chapter describes how to start using RealView® Debugger. It contains the following sections:

- *Starting RealView Debugger* on page 1-2
- *RealView Debugger directories* on page 1-6.

Note

Where RealView Debugger panes are shown in this document, they are shown as floating panes. Floating panes show the name of the pane in the title bar. Docked panes do not have a title bar, and so do not show the title.

1.1 Starting RealView Debugger

This section describes how to start the debugger. It contains the following sections:

- *Starting from Windows*
- *Starting from the command line*
- *Setting environment variables* on page 1-5.

Note

The debugger can be used with a single processor only—no multi-processing is permitted.

1.1.1 Starting from Windows

To start RealView Debugger:

1. Select **Programs** → **ARM** → **RealView Developer Kit v2.2 ...** → **RealView Debugger v1.8** from the Windows **Start** menu.

Note

If you are using the default Windows XP settings, select **All Programs**.

Support exists for Windows 2000 and Windows XP Professional only.

1.1.2 Starting from the command line

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

```
rvdebug.exe [-b|-cmd][-install=pathname][-user=name][-home[=]pathname]
[-aws=pathname|-aws=-][-init[=]connection [-exec[=]image_pathname]]
[-inc[=]pathname][-jou[=]pathname][-log[=]pathname][-stdio[=]pathname]
[-nologo]
```

where:

-aws Runs a RealView Debugger session with the specified workspace. This overrides any workspace specification that was stored when the previous session ended.

Use **-aws=-** to start without a workspace.

-b Runs a RealView Debugger session in batch mode, that is without any user interaction.

Use this with **-inc** to run a script file containing commands.

Can be replaced with `-b`.

Note

Do not use `-b` without `-inc`. If you do, RealView Debugger runs as a hidden process, and you have to use the Task Manager to terminate the `rvdebug` process on Windows.

If you use only `-inc`, the script file is run with the GUI enabled.

`-cmd` Runs the RealView Debugger in headless debugger mode, where you can use CLI commands to carry out debugging tasks. This enables you to interact with the debugger without using the RealView Debugger GUI. Also, see the *RealView Developer Kit v2.2 Command Line Reference* for details on the CLI commands.

`-exec` Specifies the image to be loaded when RealView Debugger runs. The image specification can also include section details and image arguments. You can optionally include `=`, for example `-exec=image`.

Note

You must also use `-init` when loading an image in this way.

`-home` Specifies a RealView Debugger home directory used for the debugging session. You can optionally include `=`, for example `-home=homedir`. If the specified directory does not exist, a new one is created. Where this is not specified, the default directory is used.

See *Defining the home directory on Windows* on page 1-6 for details.

`-inc` Runs a RealView Debugger session with the specified include file. You can optionally include `=`.

Use `-inc`:

- in batch mode in association with the `-bat` setting, to execute the commands contained in the file and then exit the debugger
- in command-line mode in association with the `-cmd` setting, to execute the commands contained in the file and then leave the debugger running ready to continue the debugging session
- in GUI mode on its own, to execute the commands contained in the file during a debugging session.

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

`@endpoint_connection@access_provider`

For example, `@new_arm@localhost` for RealView ICE Micro Edition (RVI-ME).

- `-install` Specifies the installation directory where this differs from the default installation. This must point to the following directory:
`install_directory\RVD\Core\...\platform`
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.
This overrides the environment variable `RVDEBUG_INSTALL`, and must be used if `RVDEBUG_INSTALL` is not set.
- `-jou` Runs a RealView Debugger session with the specified journal file open for writing. You can optionally include `=`, for example `-jou=filename`. Can be replaced with `-j`.
- `-log` Runs a RealView Debugger session with the specified log file open for writing. You can optionally include `=`, for example `-log=filename`. Can be replaced with `-l`.
- `-nologo` Runs a RealView Debugger session without displaying a splash screen.
- `-stdiolog` Runs a RealView Debugger session with the specified `STDIOlog` file open for writing. You can optionally include `=`, for example `-stdiolog=filename`. Can be replaced with `-s`.
- `-user` Specifies the user ID in the RealView Debugger home directory used for the debugging session. Where this is not specified, the default Windows login is used.
See *Defining the home directory on Windows* on page 1-6 for details.

Examples

In the following examples you must replace `...` in the specified paths with the version and build number directories of your RealView Developer Kit installation:

- To start RealView Debugger and specify a home directory, where `RVDEBUG_HOME` is not set:

`D:\ARM\RVD\Core\...\win_32-pentium\bin\rvdebug.exe`
`-home="D:\rwd_work\home\my_user_name"`
- To start RealView Debugger and specify a workspace:

```
D:\ARM\RVD\Core\...\win_32-pentium\bin\rvdebug.exe"
-aws="D:\rwd_work\home\my_user_name\friday_test.aws"
```

- To start RealView Debugger without loading a workspace:

```
D:\ARM\RVD\Core\...\win_32-pentium\bin\rvdebug.exe -aws=-
```

- To start RealView Debugger with a log file open for writing:

```
D:\ARM\RVD\Core\...\win_32-pentium\bin\rvdebug.exe
-log="D:\rwd_work\home\my_user_name\test_files\my_log.log"
```

Getting more information

To find more information on operations available from the command line, see:

- Chapter 3 *Working with Images* for details on loading images.
- Chapter 4 *Controlling Execution* for details on using log and journal files.
- Chapter 11 *Configuring Workspace Settings* for details on workspaces.

1.1.3 Setting environment variables

User-defined environment variables can be set to configure RealView Debugger.

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

```
RVDEBUG_HOME=D:\ARM\RVD\Core\...\win_32-pentium\home\my_home
```

To specify a shared location for RealView Debugger target configuration files, set:

```
RVDEBUG_SHARE=H:\ournet\devel\rvd\shared
```

1.2 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:

- *Defining the installation directory*
- *Defining the home directory on Windows*
- *Using the examples directories on page 1-7.*

1.2.1 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 define the installation directory:

1. The `-install` command line argument, where used (see *Starting from the command line* on page 1-2). This is available only on Windows systems.
2. The `RVDEBUG_INSTALL` environment variable, which is set during installation (see *Setting environment variables* on page 1-5).

1.2.2 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 define the home directory:

1. The `-home` command line argument (only available under Windows), if used (see *Starting from the command line* on page 1-2).
2. The `RVDEBUG_HOME` environment variable, if set (see *Setting environment variables* on page 1-5).
3. The `-user` command line argument, if used (see *Starting from the command line* on page 1-2). This is then used to specify the user ID in the home directory, for example set `USER=my_user_name` to specify the home directory:
`install_directory\RVD\Core\...\home\my_user_name.`
4. Your default Windows login, for example:
`install_directory\RVD\Core\...\home\WinLogID.`

If your Windows login ID contains spaces, these are converted to underscores. Any ID longer than 14 characters is automatically truncated.

Because you can choose the home directory, the installation directory and your user name, the RealView Debugger home directory is defined in this book as being in a default location:

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

Here, *user_name* is your Windows login ID. This means that your files might be stored in places other than those given in the examples.

For details on the files that are stored in the RealView Debugger home directory see the online help topic *Where is information stored?*

1.2.3 Using the examples directories

Various demonstration projects are supplied as part of the RealView Developer Kit root installation. These contain programs in the form of ARM® assembly language, C, or C++ source code files. These projects are stored in:

```
install_directory\RVDK\Examples\...\project
```

The root installation also includes demonstration projects, and associated files, for working with Flash. These are in:

```
install_directory\RVD\Core\...\flash\examples\platform
```


Chapter 2

Connecting to Targets

This chapter describes how to use the Connection Control window to view connection details. It describes how to connect to a target using the default configuration files, and how to disconnect. It contains the following sections:

- *Working with the Connection Control window* on page 2-2
- *Managing connections* on page 2-6
- *Connecting to a target* on page 2-10
- *Failing to make a connection* on page 2-16
- *Disconnecting from a target* on page 2-18.

2.1 Working with the Connection Control window

The Connection Control window, using RealView ICE Micro Edition (RVI-ME) as an example in Figure 2-1, enables you to make connections, change existing connections, and configure new ones if required.

You can display the Connection Control window in the following ways:

- Click on the blue hyperlink in the File Editor pane, if available.
- Select **Target** → **Connect to Target...** from the Code window.
- Click the **Connection Control** button, in the Connection group on the Actions toolbar. If the window is hidden, click the button twice.

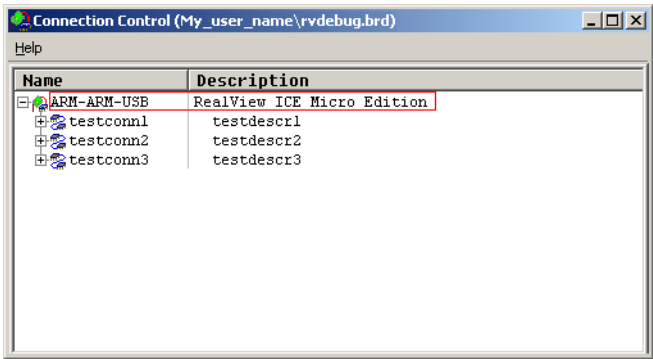


Figure 2-1 Connection Control window

Figure 2-1 shows a Connection Control window in which your connections are made.

———— **Note** ————

A software license must be present, which is obtained using the ARM® License Wizard.

RealView Developer Kit can only connect to particular ARM® architecture-based processors; each version of RVDK is designed to be used only with a particular device, or range of devices. See your *RealView Developer Kit v2.2 Getting Started Guide* for information on permitted devices.

RealView Debugger allows connection to RealView ICE/RealView Trace (RVI), and the RVI-ME supplied with RVDK.

This section describes the Connection Control window in more detail:

- *Groups in the Connection Control window*
- *Using the Connection Control window* on page 2-4
- *Changing your board file* on page 2-5.

2.1.1 Groups in the Connection Control window

Figure 2-2 shows the Connection Control window for the default board file, stored in `\home\My_user_name\rvdebug.brd`, in the root installation. In this example, one group has been expanded and a connection has been made (see *Expanding and collapsing groups* on page 2-4 for details). If you select a group, a box is drawn around it.

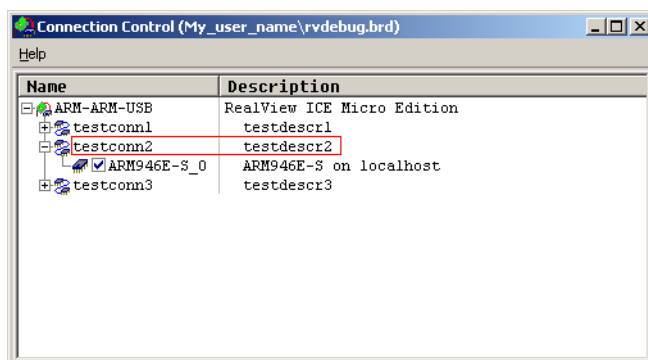


Figure 2-2 Groups in the Connection Control window

The Connection Control window is arranged in two columns or panes, Name and Description. Connection and target details are displayed in the left pane as a hierarchical tree with node controls, + and -. Connection and target details are displayed in the Name column as a hierarchy of entries:



Target vehicles

Top-level groups are supported target vehicles as specified by the target configuration settings, for example ARM-ARM-USB for RVI-ME targets, and ARM-ARM-NW for RealView ICE targets.

See *Working with target vehicles* on page 2-6 for more details on working with top-level groups.



Endpoint connections

Third-level entries show the target processors that are made available by the access provider, for example the ARM940T™ core connected using RVI-ME.



Each endpoint connection is accompanied by a check box to show the current state of the connection. When connected, this check box is checked, shown in Figure 2-2 on page 2-3. Where no connections have been made, the check boxes are unchecked.

RealView Debugger uses icons to help you identify the types of entries in the Name column.

2.1.2 Using the Connection Control window

The Connection Control window shows all the connections available to you as specified in your board file and the configuration files it references. The window title bar shows the location of the board file being used. In these examples, this is the default file, stored in `\home\My_user_name\rvdebug.brd` in the root installation.

Expanding and collapsing groups

Expand and collapse the groups in the Connection Control window by clicking on the plus sign or the minus sign at each node in the Name tree. Figure 2-2 on page 2-3 shows a Connection Control window after groups have been expanded and a connection has been made. If you select a group, a red box is drawn around it.

Context menus are available to change the way entries are displayed. To expand the top-level groups, right-click on a group, for example ARM-ARM-USB, and select **Expand Vehicles** from the context menu (see *Target Vehicle context menu options* on page 2-6). To expand the second-level entries, right-click on an entry, for example testconn2, and select **Expand**. To collapse them again, select **Collapse** from the context menu.

————— Note —————

If you are not connected, you can also expand, or collapse, entries by double-clicking.

You can connect to a target by checking the connection state check box, shown in Figure 2-2 on page 2-3. If you collapse connected (checked) entries, RealView Debugger does not complete the collapse request so that a connection is not hidden. Instead, the control is grayed out.

The Connection Control window shows available connections as defined by the target configuration settings. When you first install RealView Debugger, this is based on information about the available target processors as defined in the default configuration files. You can connect to any of the default connections, shown in the Connection Control window, without making any more changes to configuration files.

Note

You must configure certain targets before you can make your first connection, for example RVI-ME. See Chapter 14 *Configuring Custom Connections* for more details.

2.1.3 Changing your board file

If you work with a variety of targets and connections you might set up and save several board files, so that you can easily switch RealView Debugger from one to another. You can change the board file being used for the current session in two ways:

- right-click on a top-level entry in the Connection Control window, for example ARM-ARM-USB, and select the option **Select Board-File...** from the context menu (see *Target Vehicle context menu options* on page 2-6)
- change your workspace settings file to start the session with a specified board file (see Chapter 11 *Configuring Workspace Settings* for details).

Note

To ensure a consistent target view, do not change the board file when the debugger is connected to a target.

2.2 Managing connections

Use context menus in the Connection Control window to:

- manage the displayed connections
- establish a connection to your chosen debug target
- use a different board file.

There are several context menus that can be displayed, depending on the entry you select:

- *Working with target vehicles*
- *Working with access-providers* on page 2-7
- *Working with endpoint connections* on page 2-8.

2.2.1 Working with target vehicles



Right-click on a top-level entry, for example ARM-ARM-USB, to see the **Target Vehicles** context menu.

Target Vehicle context menu options

The options available from this menu enable you to manage the vehicle used to make the connection. What options are available, and how you use these options, depends on the vehicle you are using:

Collapse All Collapses the hierarchical tree to display only the top-level entries.

Expand Vehicles

Expands the hierarchical tree to display the contents of top-level groups, and any previously expanded second-level groups.

Connection Properties...

Displays the Connection Properties window to amend current configuration details or to add target configurations (see *Viewing board file properties* on page 12-5 for details).

Select Board-File...

Displays the Select Board-File to Read dialog box, where you can specify a new board file for target configuration in this session.

If you change the board file in this way, the new board is then used for all connections available with the chosen vehicle.

Note

To ensure that configuration information is maintained, do not change the active board file if:

- the Connection Properties window is open
 - you are connected to a debug target.
-

If there are any connections established, for any vehicle, the menu includes the option:

Disconnect All

Disconnects all connected targets, using the default disconnect mode, and collapses the tree to display the top-level and second-level entries.

2.2.2 Working with access-providers



Right-click on a second-level entry, for example RVI-ME, to see the **Access-provider** context menu.

Access-provider context menu options

The options available from the **Access-provider** context menu enable you to manage the debug interface used to support the connection. What options are available, and how you use these options, depends on the interface you are using:

Expand Expands the hierarchical tree to display the contents of the second-level group. This then changes to **Collapse**.

When you expand, RealView Debugger initializes and queries the selected interface. This might:

- cause a short delay while RealView Debugger queries the target to determine what connections are available
- result in error messages being displayed if a previously configured connection no longer operates because, for example, the RVI-ME configuration file cannot be located
- result in the connection configuration dialog box being displayed for the interface software.

Connection Properties...

Displays the Connection Properties window to amend current configuration details or to add target configurations (see *Viewing board file properties* on page 12-5 for details).

Configure Device Info...

Enables you to configure the debug target, for example by displaying the RVConfig dialog box where you can configure RealView ICE or RVI-ME debug targets.

For details on using these menu options to configure different targets, see Chapter 14 *Configuring Custom Connections*.

2.2.3 Working with endpoint connections

Right-click on a third-level entry, for example ARM7TDMI_0, to see the appropriate context menu:

- *Connection context menu options*
- *Disconnection context menu options* on page 2-9.

Connection context menu options

The options available from the **Connection** context menu enable you to manage the debug interface used to support the connection. What options are available, and how you use these options, depends on the interface you are using:

Connect (Defining Mode)...

Displays a List Selection dialog that lists the connect modes available for the selected target. Select the required mode and click **OK** to connect to the target. See *Connecting to a target* on page 2-10 for details on how this option is used.

Connect Connects to the selected target. See *Connecting to a target* on page 2-10 for details on how this option is used.

Connection Properties...

Displays the Connection Properties window to amend current configuration details or to add target configurations (see *Viewing board file properties* on page 12-5 for details).

Configure Device Info...

Displays the configuration dialog box for the target.

You must configure certain targets before you can make your first connection, for example RVI-ME. See Chapter 14 *Configuring Custom Connections* for more details.

Disconnection context menu options

When you select a target that is connected, the connect options on the **Connection** context menu (see *Connection context menu options* on page 2-8) change to:

Disconnect (Defining Mode)...

Displays a List Selection dialog that lists the disconnect modes available for the selected target. Select the required mode and click **OK** to disconnect from the target.

Disconnect Enables you to disconnect from the target connection.

The remaining options on the **Connection** context menu are also available depending on the target connection.

See *Disconnecting from a target* on page 2-18 for details on how these options are used.

2.3 Connecting to a target

RealView Debugger offers different ways to connect to your debug target:

- *Connections during debugging mode*
- *Using the Connection Control window*
- *Setting connect mode on page 2-11*
- *Including the connection in the workspace on page 2-15*
- *Using CLI commands on page 2-15.*

Note

You must configure certain debug targets before making your first connection, for example RVI-ME. If you try to connect to a target that has not been configured, RealView Debugger displays a dialog box to indicate that the connection has failed. See *Failing to make a connection on page 2-16* for more details.

2.3.1 Connections during debugging mode

If you already have a connection established, RealView Debugger can automatically disconnect any existing connection (see *Using auto-disconnect on page 2-19* for more details).

2.3.2 Using the Connection Control window

Select **Target** → **Connect to Target...** from the Code window main menu, or click the **Connection Control** button, to display the Connection Control window where you can connect to your debug target (see Figure 2-2 on page 2-3).

You can connect to a target in the following ways:

- Double-click on an unconnected connection entry.
- Select the check box for a required connection entry so that it is checked.
- Right-click on a connection entry and select **Connect** from the **Connection** context menu (see *Connection context menu options on page 2-8*).
- Right-click on a connection entry and select **Connect (Defining Mode)...** from the **Connection** context menu.

Note

You must use this option if you do not want the processor to be stopped when you connect to a target. For more information on defining the connection mode, see *Setting connect mode on page 2-11*.

If the chosen vehicle supports a target that has not been configured, a dialog box is displayed to indicate that the connection has failed. This enables you to configure the target first and then connect. See *Failing to make a connection* on page 2-16 for more details.

RealView Debugger connects to the specified target using the default connection mode. You can, however, specify the connection mode to use, see *Setting connect mode*.

2.3.3 Setting connect mode

When you connect to a target, RealView Debugger attempts to establish the connection using the default connect mode, that is No Reset and Stop. This mode is used when you connect from the Connection Control window in any of the following ways:

- Double-click on an unconnected connection entry.
- Select the check box for a required connection entry so that it is checked.
- Right-click on a connection entry and select **Connect** from the **Connection** context menu (see *Connection context menu options* on page 2-8).

Before connecting, RealView Debugger checks to see if a user-defined connect mode has been specified in your board file (see *User-defined connect mode* on page 2-12 for details). If such a setting is found, it overrides the default connect mode and is used instead. If there is no connect mode specified in your target configuration file, RealView Debugger proceeds to connect using the default connect mode.

However, the connect mode that is actually used depends on the target hardware, the target vehicle, and the associated interface software that manages the connection. In some cases, the interface cannot complete the connection using the mode requested, for example where your RealView ICE or RVI-ME unit configuration conflicts with the connect mode. In this case, the RealView ICE or RVI-ME unit determines the connect mode and makes the connection. Similarly, some ARM processors require a reset before you can connect, for example XScale™, whereas other cores have no such restriction. RealView Debugger displays a warning message to say that the requested connect mode cannot be honored and tells you what connection mode was used instead.

————— Note —————

This behavior also applies to disconnect mode, for example mode substitution might occur when you try to disconnect from a running target. See *Setting disconnect mode* on page 2-20 for more details.

RealView Debugger provides a mechanism to enable you to override both the default connect mode and any user-defined settings in your board file. This is useful when debugging a single processor target system, for example using RVI-ME. Therefore, you can control the way a processor starts for individual connections. To do this you must select the option **Connect (Defining Mode)...** from the **Connection** context menu in the Connection Control window (see *Connection context menu options* on page 2-8). See *Connect (Defining Mode)...* for details on how to use this option.

User-defined connect mode

Before connecting, RealView Debugger checks to see if a user-defined connect mode has been specified by the `Connect_mode` setting in the `Advanced_Information` block in your board file, or in any `.bcd` file linked to the connection. If such a setting is found, it becomes the default connect mode for this connection and is used if you connect in any of the following ways:

- Double-click on an unconnected connection entry.
- Select the check box for a required connection entry so that it is checked.
- Right-click on a connection entry and select **Connect** from the **Connection** context menu (see *Connection context menu options* on page 2-8).

For more information about setting connect mode in your board file, see:

- *About target configuration* on page 13-2 in Chapter 13 *Configuring Custom Targets*
- the example *Specifying connect and disconnect mode* on page 13-29 in Chapter 13 *Configuring Custom Targets*
- the description of the `Advanced_Information` block in Appendix B *Configuration Properties Reference*.

Connect (Defining Mode)...

To define a connection mode using the **Connect (Defining Mode)...** option:

1. Right-click on a processor connection entry, for example `ARM940T_0`, using RVI-ME, to display the **Connection** context menu (see *Connection context menu options* on page 2-8).
2. Select **Connect (Defining Mode)...** to display the Connect Mode list selection box shown in Figure 2-3 on page 2-13.

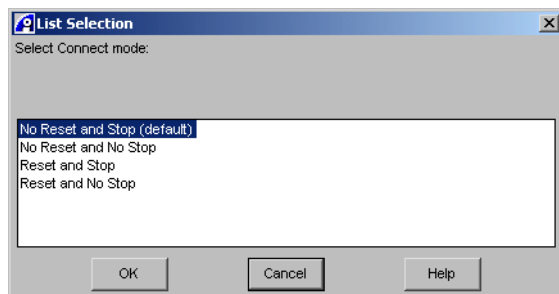


Figure 2-3 Connect Mode selection box

The state options shown depend on the interface software making the connection, but are always one or more from:

No Reset and Stop

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

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.

Highlight the required state and click **OK**.

Note

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

Default state

The selection box, shown in Figure 2-3 on page 2-13, offers a series of state options to make the connection. The options shown depend on the interface software that manages the target connection and so vary depending on the connection entry you select in the Connection Control window. In this example, the selection box contains an entry No Reset and Stop (default). This is either the:

- default connect mode chosen by the debugger (see *Setting connect mode* on page 2-11 for details)
- user-defined connect mode specified in the target configuration file (see *User-defined connect mode* on page 2-12 for details).

Any user-defined connect mode takes precedence over the default connect mode chosen by the debugger.

If you click **Cancel**, RealView Debugger establishes the connection using the connect mode shown as the default. Where this mode is not supported by your target, a different connect mode is used and you are warned about the substitution.

Note

This selection box is also displayed if you specify a prompt as your user-defined connect mode. In this case, there is no default entry (see *Using a prompt* for details).

Using a prompt

You can specify a user-defined connect mode to display a prompt to enable you to choose the connection mode for each connection. In this case, RealView Debugger displays the Connect Mode selection box shown in Figure 2-3 on page 2-13. However, where the prompt setting is defined, the state options offered are all supported by the target vehicle and so do not include a default option.

Note

If a prompt is specified in your board file, or in any .bcd file linked to the connection, it takes priority over any other user-defined connect mode setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.

For more information about using a prompt, see:

- the example *Specifying connect and disconnect mode* on page 13-29 in Chapter 13 *Configuring Custom Targets*
- the description of the Advanced_Information block in Appendix B *Configuration Properties Reference*.

2.3.4 Including the connection in the workspace

If you exit the debugger with an active connection, a record of the connection details is kept in the active workspace. The next time that workspace is active when the debugger starts, the debugger attempts to set up the previous connection again. See Chapter 11 *Configuring Workspace Settings* for more details.

2.3.5 Using CLI commands

The CONNECT and RUN commands can be used to make a connection to your debug target.

You can use the CLI CONNECT command to make numbered connections, for example:

```
connect,route 1
```

```
connect 6
```

where the connection id is used to identify the target.

You can also use the CONNECT command to specify the connect mode, for example:

```
connect,reset,halt 12
```

You can connect to remote connections in the same way.

Making named connections

You can also make named connections, for example:

```
connect @testconn1
connect @ARM946E-S_0
```

Specify the route name as defined in the board file, that is as it appears in the Connection Control window. Use the access-provider to avoid ambiguity, for example:

```
connect @ARM946E-S_0@testconn1
```

See the description of the CONNECT command in *RealView Developer Kit v2.2 Command Line Reference* for full details on connecting to targets this way.

2.4 Failing to make a connection

When you click inside a check box, in the Connection Control window, RealView Debugger might fail to connect to the chosen debug target. This might be because:

- The FLEXlm license is either not installed, or it has been installed on a different system to the one specified in the license. See the *ARM FLEXlm License Management Guide* for information about node-locked licenses.
- 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 RealView Debugger attempts to make a connection and fails, it normally displays a list selection box to offer possible actions, shown in Figure 2-4. Some endpoint connections, such as RVI-ME, might also display their own dialog boxes or messages.

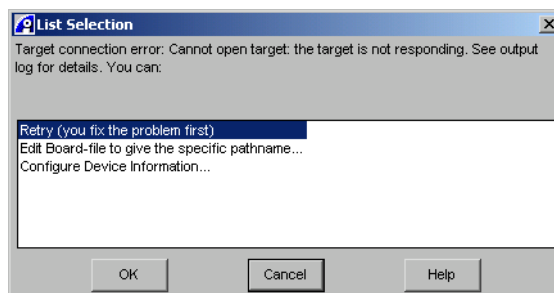


Figure 2-4 Failing to make a connection

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

Retry... If you have identified the cause of the failure and corrected it, for example you have connected a board or switched on power, you can select this option and click **OK** to connect.

Edit Board file...

You can select this option and click **OK** to close the list selection box and display the Connection Properties window where you can edit your target configuration details (see *Viewing board file properties* on page 12-5 for details).

Save the new settings and then close the Connection Properties window before trying to make the connection using the Connection Control window.

Configure Device Information...

Select this option to display the configuration dialog box for the target.

You must configure certain targets before you can make your first connection, for example RVI-ME. See Chapter 14 *Configuring Custom Connections* for more details.

Click **Cancel** to close the message box and abandon the connection.

2.4.1 Troubleshooting

This section helps you to identify and fix connection problems you might encounter:

- The selection box, shown in Figure 2-4 on page 2-16, 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.

This list provides only suggestions and might not be applicable to your debug target.

- If your working versions of configuration files are accidentally erased, or become corrupted, RealView Debugger might be unable to use them. See *Troubleshooting* on page 13-46 for information describing how to recover from this situation.
- If you are using RealView ICE or RVI-ME and see a message asking you to *Kill all other connections...*, ensure that the required connection is available before terminating other connections.

2.5 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.

This section includes:

- *Disconnect behavior in RealView Debugger*
- *Using auto-disconnect* on page 2-19
- *Using the Target menu* on page 2-19
- *Using the Connection Control window* on page 2-19
- *Setting disconnect mode* on page 2-20
- *Disconnecting by exiting* on page 2-25
- *Using the CLI* on page 2-26.

2.5.1 Disconnect behavior in RealView Debugger

Code windows are not closed on 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
- entries displayed in a Register, Memory, or Process Control pane are cleared
- entries in the Watch pane remain unchanged.

If you are working with projects, any open projects do not close if you disconnect from a debug target. Even where a project is bound to the connection, it does not close if you disconnect. However, it is unbound and if it was the current connection, its details are no longer shown in the Process Control pane.

For details on working with projects see the chapter that describes project binding in *RealView Developer Kit v2.2 Project Management User Guide*.

2.5.2 Using auto-disconnect

If you are already connected to a debug target processor, RealView Debugger automatically disconnects that connection when you make a new connection. The auto-disconnect does not occur until the new connection is successfully established.

However, if the target is running, RealView Debugger warns you and prompts for confirmation to disconnect:

- Click **Yes** to disconnect the connection with the running target.
- Click **No** to maintain the connection to the running target.

2.5.3 Using the Target menu

If you are connected to a single debug target processor, you can disconnect from the current connection, either:

- Select **Target → Disconnect** from the Code window main menu.
- Click the **Disconnect** button from the Connection group on the Actions toolbar.



This has the following results:

- the current connection is terminated immediately
- any windows attached to the current connection are unattached
- title bars and Color Boxes for all unattached windows are updated.

To close any unwanted windows, select **File → Close Window** from the main menu.

2.5.4 Using the Connection Control window

Select **Target → Connect to Target...** from the Code window main menu, or click the **Connection Control** button, to display the Connection Control window where you can disconnect from your debug target.

You can disconnect from a target in the following ways:

- Double-click on a connected entry.

- Select the check box for a required entry so that it is unchecked.
 - Right-click on a connection entry and select **Disconnect** from the **Disconnection** context menu.
 - Right-click on a connection entry and select **Disconnect (Defining Mode)...** from the **Disconnection** context menu (see *Disconnection context menu options* on page 2-9).
- This option is also available from the **Target** menu.

Note

You must use this option if you do not want the processor to be stopped when you disconnect from a target. For more information on defining the disconnection mode, see *Setting disconnect mode*.

Using any of these methods immediately terminates the connection and updates the Code window display and the active connections list. This has the following results:

- the specified connection is terminated immediately
- any windows attached to the current connection are unattached
- title bars and Color Boxes for all unattached windows are updated.

RealView Debugger disconnects from the specified target using the default disconnection mode. You can, however, specify the disconnection mode to use, see *Setting disconnect mode*.

2.5.5 Setting disconnect mode

When you disconnect from a target, RealView Debugger attempts to disconnect using the default disconnect mode, that is As-is with Debug. This mode is used when you disconnect from the Connection Control window in any of the following ways:

- Double-click on a connected connection entry.
- Select the check box for a required connection entry so that it is unchecked.
- Right-click on a connection entry and select **Disconnect** from the **Disconnection** context menu (see *Disconnection context menu options* on page 2-9).

This default disconnect mode is also used when you disconnect from the Code window using either:

- Select **Target** → **Disconnect** from the Code window main menu.
- Click the **Disconnect** button from the Connection group on the Actions toolbar.

Before disconnecting, RealView Debugger checks to see if a user-defined disconnect mode has been specified in your board file (see *User-defined disconnect mode* for details). If such a setting is found, it overrides the default disconnect mode and is used instead. If there is no disconnect mode specified in your target configuration file, RealView Debugger proceeds to disconnect using the default disconnect mode.

However, the disconnect mode that is actually used depends on the target hardware, the target vehicle, and the associated interface software that manages the connection. In some cases, the interface cannot complete the disconnection using the mode requested, for example where your RealView ICE or RVI-ME unit configuration conflicts with the disconnect mode. In this case, the RealView ICE or RVI-ME unit determines the disconnect mode and disconnects. RealView Debugger displays a warning message to say that the requested disconnect mode cannot be honored and tells you what disconnection mode was used instead.

Note

This behavior also applies to connect mode, for example mode substitution might occur when you try to connect to a running target. See *Setting connect mode* on page 2-11 for more details.

RealView Debugger provides a mechanism to enable you to override both the default disconnect mode and any user-defined settings in your board file. This is useful when debugging a single processor target system, for example to download your application and leave it running without the debugger connected. Therefore, you can control the way a processor stops when you disconnect. To do this you must select the option **Disconnect (Defining Mode)...** from the **Disconnection** context menu in the Connection Control window or from the **Target** menu from the Code window main menu. See *Disconnect (Defining Mode)...* on page 2-22 for details on how to use this option.

User-defined disconnect mode

Before disconnecting, RealView Debugger checks to see if a user-defined disconnect mode has been specified by the `Disconnect_mode` setting in the `Advanced_Information` block in your board file, or in any `.bcd` file linked to the connection. If such a setting is found, it becomes the default disconnect mode for this connection and is used if you disconnect in any of the following ways:

- Double-click on a connected connection entry.
- Select the check box for a required connection entry so that it is unchecked.
- Right-click on a connection entry and select **Disconnect** from the **Disconnection** context menu (see *Disconnection context menu options* on page 2-9).

- Select **Target** → **Disconnect** from the Code window main menu.
- Click the **Disconnect** button from the Connection group on the Actions toolbar.

For more information about setting disconnect mode in your board file, see:

- *About target configuration* on page 13-2 in Chapter 13 *Configuring Custom Targets*
- the example *Specifying connect and disconnect mode* on page 13-29 in Chapter 13 *Configuring Custom Targets*
- the description of the Advanced_Information block in Appendix B *Configuration Properties Reference*.

Disconnect (Defining Mode)...

To set the disconnect mode, either:

- select **Target** → **Disconnect (Defining Mode)...** from the Code window main menu
- select the **Disconnect (Defining Mode)...** option from the Connection Control window **Disconnection** context menu (see *Disconnection context menu options* on page 2-9).

To define a disconnect mode using the **Disconnect (Defining Mode)...** option:

1. Right-click on a connected connection entry, for example ARM940T_0, using RVI-ME, to display the **Disconnection** context menu (see *Disconnection context menu options* on page 2-9).
2. Select **Disconnect (Defining Mode)...** to display the Disconnect Mode list selection box shown in Figure 2-5.

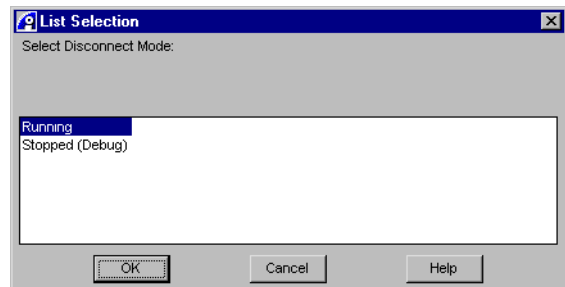


Figure 2-5 Disconnect Mode selection box

The state options shown depend on the interface software managing the connection, but are always one or more from:

As-is without Debug

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

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.

Highlight the required state and click **OK**. This has the following results:

- the current connection is disconnected
- the command is reflected in the **Cmd** tab of the Output pane
- the toolbar state group is set to Unknown
- any windows attached to the current connection are unattached
- title bars and Color Boxes for all unattached windows are updated.

The state options specify that the target is left in the current state, that is running or stopped. There is no option to stop the processor before disconnecting. To ensure that the processor is left in a particular state, you must stop (or start) the processor before disconnecting.

Remember that the disconnect mode that is actually used depends on the target hardware, the target vehicle, and the associated interface software that manages the connection. In some cases, the interface cannot complete the disconnection using the mode requested, for example where your RealView ICE or RVI-ME unit configuration conflicts with the disconnect mode. In this case, the RealView ICE or RVI-ME unit determines the disconnect mode and disconnects. RealView Debugger displays a warning message to say that the requested disconnect mode cannot be honored and tells you what mode was used instead.

————— Note —————

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

Default state

The selection box, shown in Figure 2-5 on page 2-22, offers one or more state options to disconnect from your target. The options shown depend on the interface software that manages the target connection and so vary depending on the connection entry you select in the Connection Control window. In this example, the selection box contains an entry *As-is with Debug* (default). This is either the:

- default disconnect mode chosen by the debugger (see *Setting disconnect mode* on page 2-20 for details)
- user-defined disconnect mode specified in the target configuration file (see *User-defined disconnect mode* on page 2-21 for details).

Any user-defined disconnect mode takes precedence over the default disconnect mode chosen by the debugger.

If you click **Cancel**, RealView Debugger disconnects using the disconnect mode shown as the default. Where this mode is not supported by your target, a different disconnect mode is used and you are warned about the substitution.

———— Note ————

This selection box is also displayed if you specify a prompt as your user-defined disconnect mode. In this case, there is no default entry (see *Using a prompt* for details).

Using a prompt

You can specify a user-defined disconnect mode to display a prompt to enable you to choose the disconnection mode for each connection. In this case, RealView Debugger displays the Disconnect Mode selection box shown in Figure 2-5 on page 2-22. However, where the prompt setting is defined, the state options offered are all supported by the target vehicle and so do not include a default option.

———— Note ————

If a prompt is specified in your board file, or in any .bcd file linked to the connection, it takes priority over any other user-defined disconnect mode setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.

For more information about using a prompt, see:

- the example *Specifying connect and disconnect mode* on page 13-29 in Chapter 13 *Configuring Custom Targets*
- the description of the `Advanced_Information` block in Appendix B *Configuration Properties Reference*.

2.5.6 Disconnecting by exiting

By default, exiting the debugger with a connection causes details of the connection to be stored in the current workspace. See Chapter 11 *Configuring Workspace Settings*

When you exit, the debugger prompts you to ensure you want to disconnect, as described in *Disconnection confirmation*.

When RealView Debugger starts up with a workspace that includes stored connection information, it tries to reconnect. If this fails, you are prompted for the next action, as described in *Reconnecting stored connections*.

Disconnection confirmation

If you exit the debugger with active connections, the debugger asks whether these connections can be disconnected, shown in Figure 2-6.

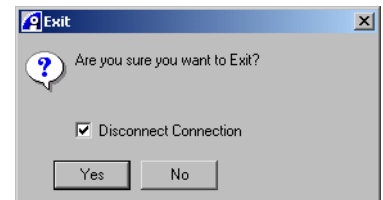


Figure 2-6 Disconnect confirmation

If you do not disconnect, the *Target Vehicle Server* (TVS) maintains the connection until another debugger session requires it. Therefore, if you load and run an image on your target, stop it, exit from the debugger without disconnecting, and then rerun the debugger, it is stopped in the same place when the debugger redisplay the connection.

If you disconnect when exiting the debugger, TVS disconnects that connection, using the disconnection mode defined in the *Advanced_Information* setting *Disconnect_mode* in the connection properties for that connection. If this leaves the TVS with no connections, it exits as well.

Reconnecting stored connections

If, when you restart the debugger, the connection stored in the workspace is no longer available, you are prompted to retry or reconfigure it, shown in Figure 2-7 on page 2-26. Select **Configure Device Information...** from the list and click **OK** to reconfigure the connection (see Chapter 14 *Configuring Custom Connections* for more details). Click **Cancel** to abort the connection attempt.

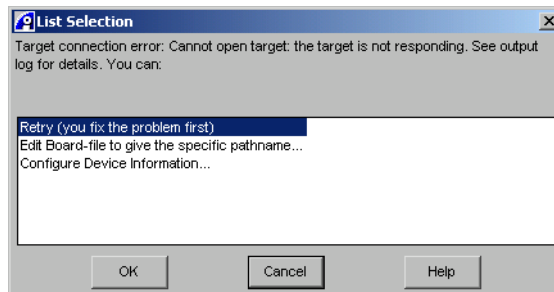


Figure 2-7 Disconnect reconfiguration or retry

2.5.7 Using the CLI

You can disconnect from a debug target using the CLI command DISCONNECT. This also enables you to specify the disconnection mode using a dialog box. See *RealView Developer Kit v2.2 Command Line Reference* for more information.

Chapter 3

Working with Images

This chapter describes how to manage images during a RealView® Debugger debugging session. It contains the following sections:

- *Loading images* on page 3-2
- *Managing images* on page 3-9
- *Working with symbols* on page 3-18
- *Working with multiple images* on page 3-19
- *Unloading and reloading images* on page 3-21.

3.1 Loading images

If you have started RealView Debugger, as described in Chapter 1 *Starting to use RealView Debugger*, you can begin to use many features of the debugger, for example editing source code and building projects. However, to begin debugging images you must connect to a suitably configured debug target.

RealView Debugger uses a *board file* to access information about the debugging environment and the debug targets available to you, for example how memory is mapped.

You can start to use RealView Debugger with the default board file installed as part of the root installation without making any more changes.

Select **Target** → **Connect to Target...** from the main menu to display the Connection Control window to make your first connection.

If you have started RealView Debugger and connected to a debug target, you can load an image to begin your debugging session. This section describes different ways to load an image to your debug target and how to monitor the loading operation:

- *Loading from a user-defined project*
- *Using the Load File to Target dialog box* on page 3-3
- *Loading from the Process Control pane* on page 3-5
- *Quick loading* on page 3-5
- *Loading from the command line* on page 3-6
- *Loading and runtime visualization* on page 3-8.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

The examples in this section 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 here.

3.1.1 Loading from a user-defined project

Where you have created a user-defined project, it is recommended that you open the project first to load and debug the associated image, or images. Opening the project enables you to access the project properties, save new settings, or make changes to the build model.

To load the image associated with an open user-defined project, click on the blue hyperlink in the File Editor pane. For the example dhrystone project, the image hyperlink is:

```
install_directory\RVDK\Examples\...\dhrystone\DebugRel\dhrystone.axf
```

————— **Note** —————

If you load an image, built as part of a user-defined project, without opening the project, this does not give you access to all the project properties because these are unknown to RealView Debugger. In this case, RealView Debugger creates an in-memory project, or uses the saved *auto-project* file (see *Working with auto-projects* on page 3-12 for details).

3.1.2 Using the Load File to Target dialog box

Select **Target → Load Image...** from the Code window main menu to load an image to a processor for execution. This displays the Load File to Target dialog box shown in Figure 3-1.

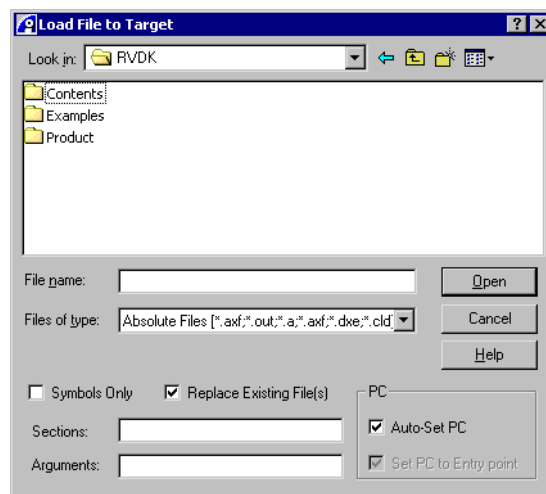


Figure 3-1 Load File to Target dialog box

This dialog box contains controls to configure the way the image is loaded for execution:

Symbols Only

By default, any object file loaded from this dialog box also loads the symbols. If you want to load only the symbols then select this check box, for example when you are working with ROM images.

If the program was initially compiled without a symbol table then you must recompile the program before loading only the symbols.

See *Working with symbols* on page 3-18 for more details.

Replace Existing File(s)

By default, loading a new image overwrites any image currently loaded to the target.

If you are working with multiple applications, use this check box to carry out separate loads of associated modules such as an RTOS and associated applications.

Sections:

Use this field to enter a comma-separated list of sections to be loaded, for example, ER_RO, ER_RW, ER_ZI.

A name entered here is then used as the argument to a LOAD command (see *Specifying the target connection and load instruction* on page 3-7).

Arguments: Use this field to enter a space-separated list of arguments to the image.

Entries in this field create an arguments list used with the LOAD command (see *Specifying the target connection and load instruction* on page 3-7).

PC

When you load an image to the debug target you can optionally set the *Program Counter (PC)*:

Auto-Set PC

Selected by default, this control defines the location of the PC when you load an image. RealView Debugger tracks the state of the other check boxes on this dialog box and sets the PC at the normal entry point, if you select the check box **Replace Existing File(s)**.

Unselect the **Auto-Set PC** check box to have control over the PC when you load an image.

Set PC to Entry point

Where selected, RealView Debugger sets the PC at the start address specified in the object module.

This is the default if you select both:

- **Auto-Set PC**
- **Replace Existing File(s).**

Unselect the **Set PC to Entry point** check box to prevent the load command setting the PC.

Note

Controls used here, for example setting the PC, might be overridden by load settings elsewhere, for example as specified in your project or target configuration settings.

3.1.3 Loading from the Process Control pane

If you have started RealView Debugger and are connected to a debug target, you can load an image for execution from the Process Control pane:

1. Select **View** → **Process Control** from the default Code window main menu to display the Process Control pane.
With no image loaded, the pane only shows details about the debug target processor and the current location of the PC.
2. Right-click on the top line, the processor entry (for example ARM7TDMI), to display the **Process** context menu.
With no image loaded, you can also display this menu from the Image entry.
3. Select **Load Image...** to display the Load File to Target dialog box.
4. Complete the entries in the dialog box, described in *Using the Load File to Target dialog box* on page 3-3, to load the required image.

3.1.4 Quick loading

With a connection established, you can load an image by dragging the appropriate executable file, with the .axf extension, and dropping it into the File Editor pane. If successful, this is the same as loading the image using the Load File to Target dialog box with the default settings (shown in Figure 3-1 on page 3-3), that is the load auto-sets the PC and overwrites any existing image on the debug target.

Note

Ensure that the target device for the current connection, shown in the Code window title bar, matches the processor type of the image you are trying to load. If they do not match the load fails.

In addition, make sure that the processor or architecture of the target matches 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.

3.1.5 Loading from the command line

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. The syntax for loading an image this way is:

```
rvdebug.exe -init=@endpoint_connection@access_provider -exec pathname
```

where *pathname* specifies the image loaded and can also include image sections to be loaded and image arguments. See *Specifying the target connection and load instruction* on page 3-7 for more details.

If the pathname includes spaces, it must be enclosed in quotation marks. For example:

```
rvdebug.exe -init=@ARM946E-S_0@testconn1 -exec "C:\rwd_work\my images\my_image.axf"
```

This starts RealView Debugger, connects to RVI-ME, and issues 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. For details of the LOAD command, see *RealView Developer Kit v2.2 Command Line Reference*.

Note

For details on running RealView Debugger from the command line see Chapter 1 *Starting to use RealView Debugger*.

Connection prompt on load failure

Starting RealView Debugger in this way connects to the target, loads the specified image to that target, and updates the Code window. This is the same as *Using the Load File to Target dialog box* on page 3-3.

If you do not specify a target connection when starting RealView Debugger from the command line, or the specified connection fails, the debugger displays a prompt box, shown in Figure 3-2 on page 3-7, for you to complete the connection.

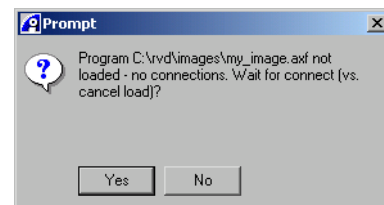


Figure 3-2 Connection prompt

Click:

- Yes** Causes the debugger to wait until you successfully connect to your debug target. The image is then loaded to the connected target.
- No** Starts the debugger but cancels the image loading operation.

Specifying the target connection and load instruction

To load an image from the command line, you must also specify a target connection. You can also pass arguments to the image and specify any image sections to be loaded. The syntax is as follows:

```
rvdebug.exe -init=@endpoint_connection@access_provider -exec
image.axf;[sections];[arg1 arg2 ...]
```

where:

endpoint_connection

Specifies the endpoint connection name.

access_provider

Specifies the access provider name for the connection.

image.axf

Specifies the image to be loaded.

sections

Specifies the 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 —————

Spaces must not be included between the argument and the qualifier. Where an arguments list is given, quotation marks must be used.

For details on specifying sections and arguments, see the LOAD command in *RealView Developer Kit v2.2 Command Line Reference*.

Examples

The following examples show how to use the `-init` and `-exec` arguments:

- To establish a connection to RVI-ME, and load an image with sections `ER_R0` and `ER_RW` and the argument `5000`:

```
rvdebug.exe -init=@ARM946E-S_0@testconn1 -exec
"C:\rvd\images\my_image.axf;ER_R0,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 RVI-ME, and load an image without specific sections and without arguments:

```
rvdebug.exe -init=@ARM946E-S_0@testconn1 -exec C:\rvd\images\my_image.axf
```

3.1.6 Loading and 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:W

- here 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.
- 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.

3.2 Managing images

This section describes how to manage your application files in the Code window. It contains the following sections:

- *Viewing image details in the Code window*
- *Viewing image details in the Process Control pane on page 3-10*
- *Working with auto-projects on page 3-12*
- *Working with user-defined projects on page 3-16.*

The examples in this section 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 here.

3.2.1 Viewing image details in the Code window

If an image is successfully loaded to the target processor, the Code window is updated, shown in Figure 3-3.

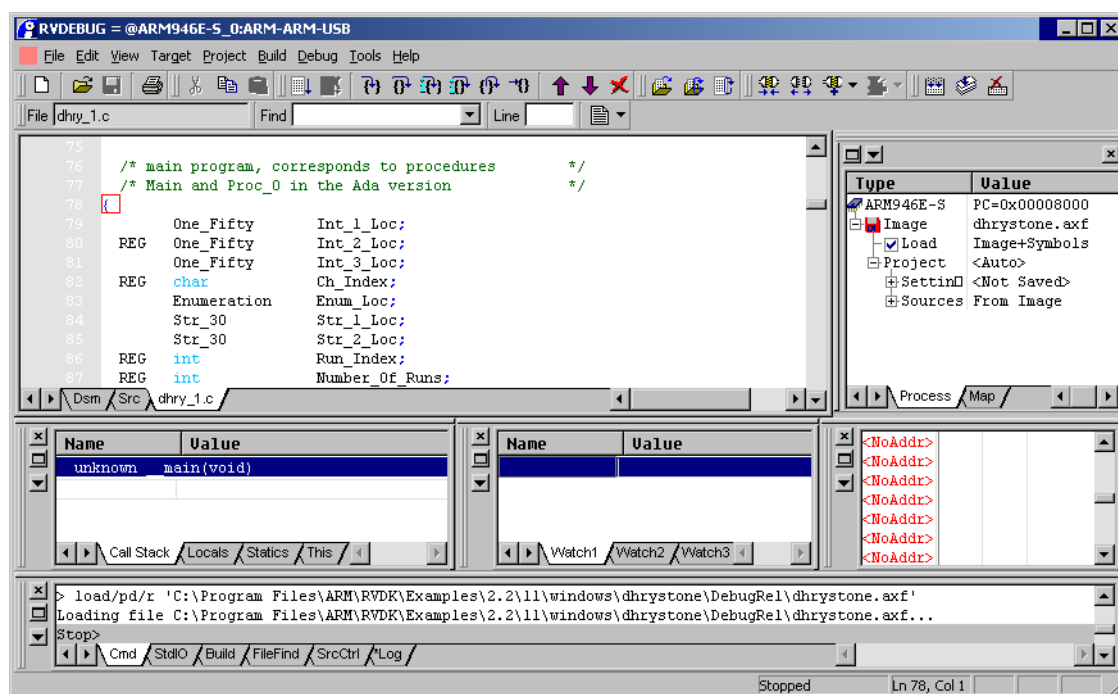


Figure 3-3 Code window with image loaded

Note

In this Code window **Text Coloring** is enabled by default and line numbering is turned on by selecting **Edit** → **Advanced** → **Show Line Numbers**.

RealView Debugger updates the panes with information about the new image, where known. Because you have not started debugging, other panes are empty.

When you load an image with symbols, as here, RealView Debugger searches for the corresponding source file associated with the current execution context (usually that containing `main()`) and displays the file as a tab in the File Editor pane. The **Src** tab acts like a button to display the current source if it is available. In this example, click on the **Src** tab to display the source-level code view.

The image was loaded with the **Auto-set PC** option set and so execution control is located at the default entry point. This is indicated by a box at line 78, colored red by default.

Click on the **Dsm** tab to show the disassembly-level view.

See *Code views* on page 4-3 for more details on using these tabs.

3.2.2 Viewing image details in the Process Control pane

By default, the side pane to the right of the File Editor pane is the Process Control pane. However, if you no longer have the Process Control pane visible, select **View** → **Process Control** to display it as a floating pane, shown in Figure 3-4.

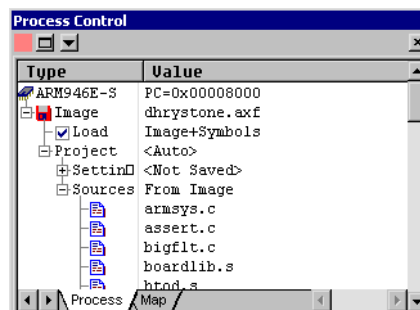


Figure 3-4 Image details in the Process Control pane

The Process Control pane contains tabs:

Process Displays details of the target processor. See *Working with processes* on page 3-11 for details.

Map Displays the memory mapping for the target processor, or the current process. You can also temporarily change the map settings if required.
See Chapter 6 *Memory Mapping* for details on using this tab.

The tabs displayed in the Process Control pane 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 displayed. See Chapter 15 *RTOS Support*.

Working with processes

The Process Control pane shows details about each connection known to RealView Debugger. If you are debugging a single process application, use the **Process** tab to see the processor details, project details, and information about any image(s) loaded onto the debug target, for example:

- image name
- image resources, including DLLs
- how the image was loaded (that is, image with symbols, or symbols only)
- associated files.

Use context menus in the **Process** tab to:

- reset your target processor (where supported)
- load, unload, and reload images, and refresh symbols
- manage settings for auto-projects and user-defined projects
- open a specified source file.

In the example in Figure 3-4 on page 3-10, you can see the 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 location 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.

- Project** Shows that the project associated with the connection is either a real, user-defined project file (shown by the project name) or an auto-project (shown by <Auto>).
- Settings** Shows where project settings are stored. These might be from a disk file (shown by <Saved>) or from an in-memory auto-project (shown by <Not Saved>).
- Sources** These are either the sources making up the project, sources extracted from the makefile used in the build, or sources from the loaded image.
- Depending on the type of project, right-click on this entry to display a context menu to specify how sources are collected.

Working with source files

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

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

Getting more information about an entry in the Process tab

Right-click on an entry in the **Process** tab to see the context menu associated with that entry. Select **Properties** to see a text description of the item under the cursor.

————— Note —————

The options available from the context menu depend on which entry is selected and the current state of the process or processor.

3.2.3 Working with auto-projects

An auto-project is an in-memory Custom project. The project settings are obtained from the image, where known. However, because no build details are known, the image cannot be rebuilt from the auto-project. See the *RealView Developer Kit v2.2 Project Management User Guide* for more details about auto-projects and Custom projects.

When you load an image directly to a debug target, RealView Debugger checks to see if an auto-project file exists for the image in the same location. Where an auto-project exists, RealView Debugger opens it and then uses it to load the specified image. Where no auto-project exists, RealView Debugger creates an in-memory auto-project to use in this session.

If, for example, you load the image `dhrystone.axf`, in `install_directory\RVDK\Examples\...\dhrystone\DebugRel`, RealView Debugger looks for the corresponding auto-project file `dhrystone.axf.apr`, in the same location. Where no auto-project exists, RealView Debugger creates an in-memory auto-project, named `dhrystone`. The **Process** tab is then updated with the project details, shown in Figure 3-5.

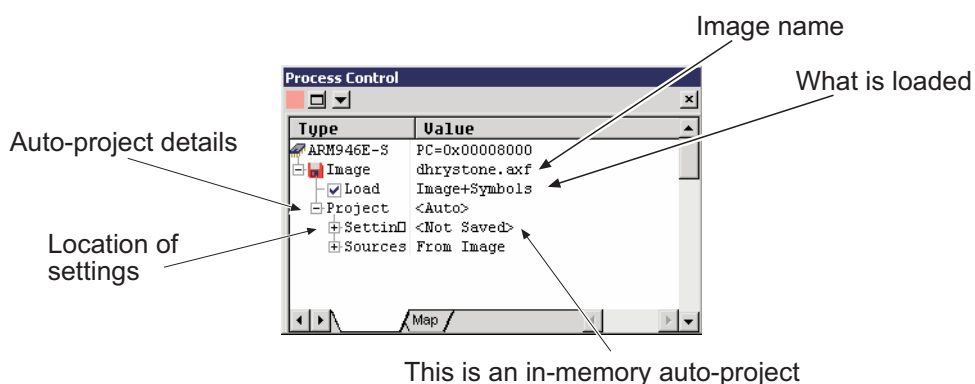


Figure 3-5 Auto-projects in the Process Control pane

RealView Debugger gives you the option to save the in-memory settings to a file to use next time the image is loaded or as the basis of a new user-defined project.

Viewing project settings

You can view the settings for the in-memory auto-project in the same way as for a user-defined project:

1. Right-click on either the **Project** entry or the **Settings** entry, to display the **Project** context menu.
2. Select **Project Properties...** to display the Project Properties window where you can view the project settings. These are derived from the image details or created using defaults by RealView Debugger.
3. Select **File → Close Window** to close the Project Properties window without making any changes.

Note

Do not leave the Project Properties window open when you are working with the debugger. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open (see *Searching for source files* on page 4-17 for details).

Changing project settings

You can change load settings for an image where you do not have a user-defined project by defining actions in the auto-project and then the saving the file for use next time the image loads. You can specify commands to execute when the project opens and/or closes, or runtime controls that define the image environment.

Note

Changing auto-project settings might not take effect until the next time the image is loaded and executed. Reload an image to implement any new settings.

You can change settings for the in-memory auto-project in the same way as for a user-defined project:

1. Right-click on the Project entry, to display the **Project** context menu.
2. Select **Project Properties...** to display the Project Properties window.
3. Expand the PROJECT group to see the project settings, shown in Figure 3-6.

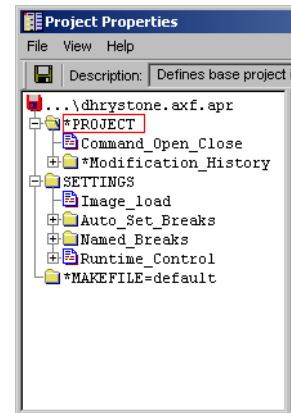


Figure 3-6 Changing auto-project settings

Here you can see the Command_Open_Close group and other project settings.

4. Expand the **SETTINGS** group to see the image settings.
Figure 3-6 on page 3-14 shows the **Image_load** group and other image settings, such as breakpoints and runtime controls.
5. Right-click on the **Image_load** group and select **Explore** to see the group contents in the right pane.
6. Right-click on the **Set_pc** entry and select **never** from the options.
7. Select **File** → **Save and Close** to save your changes and close the Project Properties window.

To return the setting to the default:

1. Display the Project Properties window.
2. Right-click on the entry to display the context menu.
3. Select **Reset to Default** to restore the setting.
4. Select **File** → **Save and Close** to save your changes and close the Project Properties window. The changes are saved to the file `dhrystone.axf.apr`.

Note

There are full descriptions of the entries in the Project Properties window, with details of the available options, in the RealView Debugger online help topic *Changing Project Properties*.

Saving project settings from the Process Control pane

You can save the default project settings for an auto-project from the Process Control pane, so that the settings are used when you next load the image:

1. In the Process Control pane **Process** tab, right-click on the Project **<Auto>** entry.
2. Select **Save** from the **Project** context menu to save the setting to the file `dhrystone.axf.apr`.

Note

This option is not available if an auto-project file already exists.

Deleting a saved auto-project file

You can delete a saved auto-project so that the file is removed from your disk:

1. Right-click on the Project entry, to display the **Project** context menu.
2. Select **Delete Auto-Project File** to remove the saved file.

Closing auto-projects

To close an auto-project, right-click on the Project <Auto> entry in the **Process** tab and select **Close** from the **Project** context menu. If you close the auto-project associated with a loaded image, this immediately unloads the image and removes all image details from the debugger. If you close an auto-project, RealView Debugger executes any close commands associated with the project.

Note

You can also use the **Project** menu from the Code window main menu to close auto-projects.

See the chapter that describes working with multiple projects in *RealView Developer Kit v2.2 Project Management User Guide* for more details on working with auto-projects.

3.2.4 Working with user-defined projects

With a user-defined project open, right-click on the Project entry to display the **Project** context menu.

This menu enables you to view details of your project, make changes to project settings, and to perform selected components of the build model following changes to project files.

See the description of the **Project** menu in the chapter that describes working with projects in *RealView Developer Kit v2.2 Project Management User Guide* for full details on these options.

Closing user-defined projects

To close a user-defined project where the associated image is loaded, right-click on the Project entry in the **Process** tab and select **Close** from the **Project** context menu.

RealView Debugger gives you the option to unload the image when the project closes.

If you choose to unload the image, RealView Debugger completes the operation, closes the project, and then executes any close operations associated with the project.

If you do not unload the image, the debugger:

1. Closes the user-defined project.
2. Executes any close commands associated with the project.
3. Either:
 - opens the saved auto-project file, where one exists for the image
 - creates an in-memory project where no saved auto-project exists.

It is not necessary to reload the image following these actions.

Note

You can also use the **Project** menu from the Code window main menu to close user-defined projects in the same way.

See the chapter that describes working with projects in *RealView Developer Kit v2.2 Project Management User Guide* for more details on closing your projects.

3.3 Working with symbols

An executable image contains symbolic references, such as function and variable names, in addition to the program code and data.

If you select the **Symbols Only** check box, on the Load File to Target dialog box (see Figure 3-1 on page 3-3), 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 RTOS-enabled target.

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, depending on the number of images loaded:

- select **Target** → **Refresh Symbols** from the Code window main menu
- use the Process Control pane:
 - right-click on the Image entry to display the **Image** context menu
 - select **Refresh Symbols** from the available options.

Note

When an image is loaded with symbols, the symbol table is recreated. This automatically deletes any macros because these are stored in the symbol table.

3.4 Working with multiple images

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 RTOS at the same time.

To load two images to the same debug target:

1. Load the first image, for example `hello.axf`, in any of the load methods. This can be either a standalone image, or an image that is associated with a user-defined project.
 2. Load a second image to the same target, for example `demo.axf`. The two images must not overlap in memory. You must load the second image as follows:
 - a. Select **Target** → **Load Image...** from the main menu to display the Load File to Target dialog box (see *Using the Load File to Target dialog box* on page 3-3).
 - b. Unselect the **Replace Existing File(s)** check box, to prevent this image replacing the first image.
 - c. Unselect the **Auto-Set PC** check box, then unselect the **Set PC to Entry point** check box. This makes sure 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.
- d. Click **Open**.
 3. Select **View** → **Process Control** from the default Code window main menu to display the Process Control pane.
 4. Expand the display to see the process details, shown in Figure 3-7.

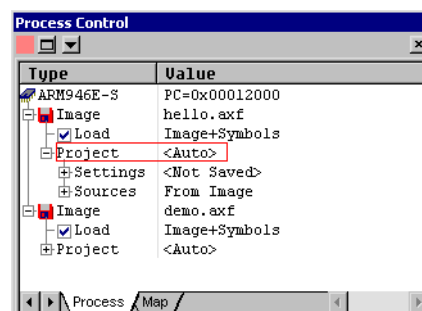


Figure 3-7 Multiple images in the Process Control pane

In this example, RealView Debugger:

- creates an in-memory auto-project for `hello.axf`
- binds `hello.axf.apr` to the current connection (using *default binding*)
- creates an in-memory auto-project for `demo.axf`
- does not bind `demo.axf.apr`, because there is no connection available
- updates the Code window title bar to show the active project (`hello`).

For information on projects and project binding see *RealView Developer Kit v2.2 Project Management User Guide*.

Because neither 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 move the PC manually to start debugging or reload the image that you want to test. For information on changing the PC see:

- Chapter 4 *Controlling Execution* for details on setting scope.
- Chapter 7 *Working with Debug Views* for details on changing register entries.

———— **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.

3.5 Unloading and reloading images

This section describes how to unload and reload images without ending your debugging session. It contains the following sections:

- *Resetting your target processor*
- *Unloading an image*
- *Replacing the currently loaded image* on page 3-22
- *Reloading an image* on page 3-22.

3.5.1 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.

If the processor chosen for reset has an image loaded then this might be unloaded on reset. The image can be reloaded as described in *Reloading an image* on page 3-22.

To reset a processor:

1. Select **View** → **Process Control** from the default Code window main menu to display the Process Control pane.
2. Right-click on the Process entry to display the **Process** context menu.
3. Select **Reset Target Processor** to perform the reset.

3.5.2 Unloading an image

Use the Process Control pane if you do want to unload an image explicitly:

1. Right-click on the Image entry to display the **Image** context menu.
2. Select **Unload** from the available options.

This is the same as clicking on the **Load** check box to unload the image.

If there are any source file tabs currently displayed in the File Editor pane, the **Src** tab is brought to the top automatically when you unload because there is no known context. The open files remain available for editing, if required. See *Code views* on page 4-3 for more details on using these tabs.

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 and the associated auto-project, or user-defined project used to load the image, is maintained.

Note

Unloading an image does not close the in-memory auto-project, associated with the image, or save any changes to the auto-project. This enables you to modify these settings, and save, ready for the next time you load (or reload) this image. However, if you close the auto-project explicitly, RealView Debugger performs an image unload.

Deleting image details

To remove all details about an image after you have unloaded it, right-click on the Image entry in the Process Control pane and select **Delete Entry** from the context menu. If there is an auto-project associated with the image, this closes. If you opened a user-defined project to load the image, this does not close.

Disconnecting with an image loaded

If you disconnect with an image loaded, this removes debug information from RealView Debugger and so clears pane contents from your Code window. This does not close the user-defined project used to load the image, or any auto-project associated with the image. You can reload the image if you reconnect.

However, if you close the project explicitly, you must load the image again after you reconnect because all image details are removed.

3.5.3 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, ensure that the **Replace Existing File(s)** check box is checked on the Load File to Target dialog box, shown in Figure 3-1 on page 3-3. This automatically removes all details about the first image from RealView Debugger.

3.5.4 Reloading an image

During your debugging session you might have to edit your source code and then recompile. Following these changes, you must either:

- load the previously unloaded image
- reload the image.

Reloading refreshes any window displays and updates debugger resources.

To reload an image:

- Select **Target → Reload Image to Target** from the Code window to reload an updated image to your debug target.

An image that has been unloaded cannot be reloaded using this option. Instead you must load the image again using **Target → Load Image....**

- Click the **Reload Image** button on the Image toolbar.
- Select the **Load** check box in the Process Control pane.

This is the same as double-clicking on the Load entry.

RealView Debugger uses auto-project settings to load an image and these are automatically used when you reload the image, or when you select it from the Recent Images list.

If you used the Load File to Target dialog box to enter a space-separated list of arguments to the image or comma-separated list of sections, these are not used when you reload (see *Using the Load File to Target dialog box* on page 3-3 for details). To reuse the arguments and sections either:

- Select **Target → Recent Images** to reload the image from the Recent Images list.
- Click the **Set PC to Entry** button on the Image toolbar to submit a RESTART command.



Chapter 4

Controlling Execution

There are several ways to control program execution from the RealView® Debugger Code window. These are described in the following sections:

- *Submitting commands* on page 4-2
- *Defining execution context* on page 4-3
- *Using execution controls* on page 4-7
- *Working with the Debug menu* on page 4-10
- *Automating debugging operations* on page 4-13
- *Searching for source files* on page 4-17.

4.1 Submitting commands

You can submit commands to RealView Debugger to control debugging behavior in several ways, for example by choosing from the **Debug** menu, or by clicking on a control on the Debug toolbar, or by typing a command-line instruction at the prompt.

If an application is currently executing, RealView Debugger uses a command queue to handle those 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.

If a command is currently executing and you request another action, the new command is added to the queue and pends until it can be executed. A warning message is displayed, in the Output pane, to explain what is happening to the new command, for example:

```
> go  
Command pended until execution stops. Use 'Cancel' to purge.
```

If the application is still running, any additional commands you enter are then added to the queue behind this pending command.

The following notes apply to the command queue:

- All commands are added to the queue if they cannot be executed immediately.
- RealView Debugger appends unknown commands, and so possibly invalid commands, to the command queue.
- Breakpoints are set where possible, otherwise these commands also pend until they can be executed.
- Known invalid commands, for example those that do not start with a letter, are not added to the queue.



Click the **Cancel** button on the Debug toolbar to clear, or purge, the most recent pending command.

4.2 Defining execution context

RealView Debugger enables you to define the current execution context, and to change this if required. You do this using:

- *Code views*
- *Defining scope and context.*

4.2.1 Code views

Use the File Editor pane to view source code during your debugging session. In the example shown in Figure 3-3 on page 3-9, the File Editor pane contains three tabs:

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

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

The **Src** tab acts like a button to display the current source if it is available. If the source is not available, RealView Debugger displays a No source message in the **Src** tab window.

If you click on a tab, for example the **Src** tab, the source file containing the current context is displayed. If the source file is not currently open, RealView Debugger opens the file. If the statement where the PC is located is in view, a red box is drawn around that statement to highlight it in the code view.

4.2.2 Defining scope and context

RealView Debugger uses *scope* to determine the value of a symbol. Scope shows how RealView Debugger accesses variables and finds symbols in expressions. The scope determines the execution *context* and defines how local variables are accessed. Any symbol value available to a C or C++ program at the current PC is also available to RealView Debugger.

When your program is executing, the PC stores the address of the current execution point. 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. In this case, RealView Debugger shows the last calling function that had valid source in the **Src** tab.

RealView Debugger uses a red box to highlight the location of the PC where this is visible in the selected code view. The PC is only visible in an execution tab, that is the **Src** or **Dsm** tab. However, if you *force* scope to a different location then this is identified by a filled blue arrow and the red box highlights the current context. This might not be the true location of the PC. See *Forcing scope* for details.

When RealView Debugger first loads an image, and assuming that you do not force scope, the File Editor pane contains tabs showing program execution (described in *Code views* on page 4-3):

- the **Src** tab shows the current context, that is the location of the PC at the entry point
- the **Dsm** tab displays disassembled code with intermixed C/C++ source lines and, if available, the location of the PC.

Locating the Program Counter

You can locate the PC using the following methods:


- Select **Debug** → **Show Line at PC** to display the current location of the PC in the Output pane. The line of code at the PC is also shown in the current view of the File Editor pane.
- Right-click on the background in the current view of the File Editor pane, and select **Scope to PC** from the context menu. The line of code at the PC is shown in the current view of the File Editor pane, and the current context of the PC is displayed in the Output pane.
- Select **Debug** → **Show Context of PC** to display the current context of the PC in the Output pane.
- Right-click on the background in the current view of the File Editor pane, and select **Scope Context** from the context menu. The current context of the PC is displayed in the Output pane.

Forcing scope


Scope is forced when it is not set by the PC. To force the scope:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Edit** → **Advanced** → **Show Line Numbers** to display line numbers.
This is not necessary but might help you to follow the examples.
3. Click on the **Src** tab to view the source file `dhry_1.c`.

The PC is at the entry point at line 78, marked with a red box.

4. Right-click at a location (line or address) in your execution view, for example line 149 in the source file `dhry_1.c`.
5. Select **Scope To Here** from the context menu.
The forced scope is identified by a filled blue pointer at the chosen location. This moves the red box to highlight the current context.
6. Select **Debug → Show Context of PC** to see the location of the PC in the Output pane.
-  7. Click **High-level Step Into** to step into the program on the Debug toolbar.
8. Select **Debug → Show Context of PC** to see the location of the PC.

To reset the PC to the entry point, either:

- Select **Debug → Set PC to Entry Point** from the Code window main menu.
-  • Click **Set PC to Entry** on the Image toolbar.

If you reset the PC to the entry point, this issues a RESTART command but does not reset the values of variables, reset the *Stack Pointer* (SP), or clear breakpoints.

Note

You can also scope to various parts of your application with the Data Navigator pane. See *Using the Data Navigator pane* on page 9-5 for details on how to use this pane.

Setting disassembly format

When working in disassembly view, you can temporarily choose the display format for your code view:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Click on the **Dsm** tab.
3. Right-click on whitespace inside the File Editor window to see the **Disassembly View** context menu.
4. Select **Set Disassembly Format...** from the context menu to see the Disassembly Mode selection box, shown in Figure 4-1 on page 4-6.

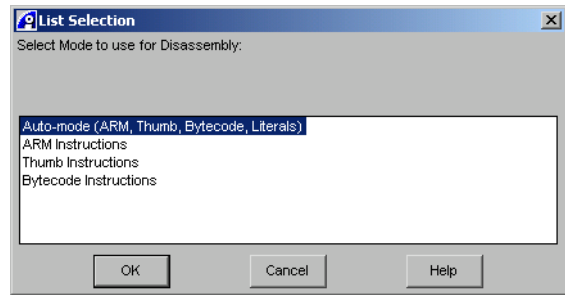


Figure 4-1 Disassembly Mode selection box

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.

5. 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. For details, see the description of the DEBUGGER group (Disassembler settings) in Appendix A *Workspace Settings Reference*.

4.3 Using execution controls

The **Execution** group, on the Debug toolbar, contains buttons to control the execution of the image loaded to your debug target. This section describes how to access execution controls:

- *Using the Execution group*
- *Using shortcuts on page 4-9.*

———— Note ————

In the following examples, loading the image `dhrystone.axf` places the PC at the entry point and not at the start of `main()`. RealView Debugger indicates the current context by highlighting the location of the PC with a red box. Any subsequent stepping instructions are based on this starting point.

4.3.1 Using the Execution group

The Execution group contains:



Run

Click this button to start from the current location of the PC and run until the program:

- ends
- encounters an error condition
- reaches a breakpoint
- reaches a halt condition.

This option can also be used to resume program execution after stopping.



Stop Execution

Click this button to stop the program currently executing on the target processor.



High-level Step Into

Click this button to step by lines of source code until the PC is located in another function or context. Normally, click this button to step up by one call level, but, depending on the function stepped into, it might cause the program to execute until it reaches source code. If the source line makes a function call, RealView Debugger steps into the function, unless there is no source code available for this function.

To see an example using this option:

1. Connect to your target and load an image, for example `dhrystone.axf`.

2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Select **Edit** → **Advanced** → **Show Line Numbers** to display line numbers.
This is not necessary but might help you to follow the examples.
4. Display line 78.
5. Set a default breakpoint by double-clicking on the line number in the left margin.
6. View the Output pane message:
`bi \DHRY_1\#78:2`
7. Click **Run** and the program begins execution and runs up to the breakpoint.
The Output pane shows where execution stops.
8. Click **High-level Step Into** once. A red box shows the location of the PC at line 91.
9. Click **High-level Step Into** several times to move through the lines of source code.

In this example, RealView Debugger completes two high-level steps at the start. Therefore, the first high-level step takes the PC from the entry point to `main()`, and the second steps to after the function prolog.

———— **Note** ————

You can get to `main()` using **Run** (as described in this example) or a single step. The single step executes (from the high-level code) up to `main()`.



High-level Step Over

Click this button to step by lines of source code over function calls.

If the source line makes a call to a function, RealView Debugger executes the function completely before stopping, assuming that there is no stopping condition in the function call, for example a breakpoint.



Low-level Step Into

Click this button to step by instructions into functions. RealView Debugger executes the assembly language instruction at the current location of the PC.



Low-level Step Over

Click this button to step by instructions over function calls. RealView Debugger executes the assembly instruction at the current location of the PC.

If the assembly instruction makes a call to a function, RealView Debugger executes the function completely before stopping, assuming that there is no stopping condition in the function call, for example a breakpoint.



Run until Return

Click this button to start from the current PC in the current function and to run until it returns to the calling function.



Run to Cursor

Click this button to start from the current location of the PC and to run until it reaches a temporary breakpoint, defined by the cursor position.

Note

In source code, repeated uses of low-level step commands might be necessary to complete execution if the source line contains multiple ARM® instructions. Low-level step instructions, **Low-level Step Into** or **Low-level Step Over**, complete one assembly instruction at a time. This means that a call to a subroutine is treated as one instruction if you execute a step over.

4.3.2 Using shortcuts

The controls in the Debug toolbar are independent of the current code view, that is clicking a button carries out the specified action regardless of whether you are in source-level view (the **Src** tab) or disassembly-level view (the **Dsm** tab). The Status line at the bottom of the Code window gives a description of the action available from a button.

Keyboard shortcuts are available, shown in the **Debug** menu, depending on the current code view. These are summarized in Table 4-1.

Table 4-1 Keyboard shortcuts

Code view	Function key	Action
Source	F10	Step by line over functions
Source	F11	Step by line into functions
Disassembly	F10	Step by instruction over functions
Disassembly	F11	Step by instruction into functions

4.4 Working with the Debug menu

Debugging your application programs relies on being able to control the execution of your code on the debug target. You must then be able to examine the contents of memory, registers, or variables, possibly continue execution one instruction at a time, or specify other actions to examine in detail the execution history. The Code window main menu includes the **Debug** menu, which provides a starting point for many debugging operations.

4.4.1 Using the Debug menu

The **Debug** menu offers the options:

Run With an image loaded to the target processor, select this option to run the program, starting from the current location of the PC. This issues the G0 command (see *RealView Developer Kit v2.2 Command Line Reference* for details).

Stop Execution

Select this option to stop the program currently executing on the target processor.

Reset Target Processor

Performs a processor reset operation on the current connection. This issues the RESET command (see *RealView Developer Kit v2.2 Command Line Reference* for details).

The reset might be hard or soft depending on the processor type, see your processor hardware documentation for details.

Simulating a processor reset does not reset variables to their defaults because memory is not re-initialized. The PC is reset.

Set PC to Entry Point

Sets the PC to the entry point of the image. This issues the RESTART command (see *RealView Developer Kit v2.2 Command Line Reference* for details). This means that you can run the image again, without having to do an image reload. Use this if you want to preserve:

- any breakpoints that you have set
- any sections or arguments that you loaded with the image.

Step Into Steps execution, by lines of source code or assembler instructions, into functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab. This issues the STEP command (see *RealView Developer Kit v2.2 Command Line Reference* for details).

Step Over (next)

Steps execution, by lines of source code or assembler instructions, over functions. This behavior depends on the current code view, that is whether you have selected the **Src** tab or the **Dsm** tab. This issues the STEPOVER command (see *RealView Developer Kit v2.2 Command Line Reference* for details).

Run until Return

Select this option to run from the current PC until control returns from the current function. This issues the G0 @1 command (see *RealView Developer Kit v2.2 Command Line Reference* for details of the G0 command).

Selecting this option stops execution at the assembler instruction immediately after the return, and not the next statement. If you select the **Src** tab, this has the effect that the red box indicating the position of the PC might be still located at the function call.

Run to Cursor

With an image loaded, you can scroll down the source or disassembly listing and position the cursor on a specific line. Select **Run to Cursor** to execute the program up to the temporary breakpoint at the cursor, assuming that no halting condition occurs first. This issues either a G0 *source_line* or G0 *address* command depending on the current code view (see *RealView Developer Kit v2.2 Command Line Reference* for details of the G0 command).

If you select the **Src** tab, the line marked by the cursor is enclosed in a red box indicating the position of the PC.

Click **Run**, or use a **Step** control, to resume execution.

Step Until Condition...

Displays a prompt where you can specify a condition, in the form of an expression. When the condition is met, that is the condition is nonzero, execution halts.

Click **OK** to run in single steps from the current location of the PC. RealView Debugger checks the condition after each step.

Using this option causes execution to slow down and might result in errors because of timing issues.

Breakpoints

This provides access to the **Breakpoints** menu to use breakpoints in your code. You can set unconditional software breakpoints, conditional software breakpoints from the **Conditional** submenu, and hardware breakpoints from the **Hardware** submenu. You must use a debug target that supports hardware breakpoints. See Chapter 5 *Working with Breakpoints* for more details on using these menu options.

Tracepoints

This enables you to set tracepoints so that trace data can be captured.

Memory/Register Operations

This enables you to examine memory and register contents during execution and to edit these interactively. In this way you have complete control over the way your application program runs. See Chapter 8 *Reading and Writing Memory, Registers and Flash* for more details on using these menu options.

Processor Exceptions

See *Specifying processor exceptions (global breakpoints)* on page 5-31.

Show Line at PC

Select this option to report the current module and procedure scope. This issues the SCOPE command (see *RealView Developer Kit v2.2 Command Line Reference* for details).

Show Context of PC

Select this option to report the current context showing the current root, module, procedure, and line. This issues the CONTEXT command (see *RealView Developer Kit v2.2 Command Line Reference* for details).

Toggle Source/Disassembly

Select this option to toggle between the source-level view and the disassembly-level view in the File Editor pane.

Set Source Search Path...

Use this to display the Project Properties window, where you can set the path to enable RealView Debugger to find sources where the compiler or assembler does not pass source paths (see *Searching for source files* on page 4-17 for details).

4.5 Automating debugging operations

The **Tools** menu includes options that enable you to automate debugger operations. You can use include files that contain CLI commands to perform the various debugging operations. You can create these files manually using a text editor, or by using the logging facilities of RealView Debugger:

- *Creating log and journal files*
- *Using include files* on page 4-16.

4.5.1 Creating log and journal files

RealView Debugger writes log and journal output to a file saved in a specified location. If the file does not exist, RealView Debugger creates it. Where a file exists, RealView Debugger gives you the option to append entries to the file, or to overwrite the current contents.

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 for writing.
- STDIO files** The *STDIOlog* log file enables you to keep a record of messages 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 for writing.
- 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 pane
 - messages from your application.
- By default, journal files have the extension `.jou`, but you can use any extension for writing.

Creating a log file for use as an include file

To create an log file for use as an include file, 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.
2. Specify the pathname of the new log file, or locate a file created previously, for example `\home\my_user_name\my_log.log`.
3. Click **Save** to confirm the settings and close the dialog box.
4. If the specified log file already exists, RealView Debugger displays the File Exists prompt.

This gives you the option to append or replace. 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 now 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 4-2.

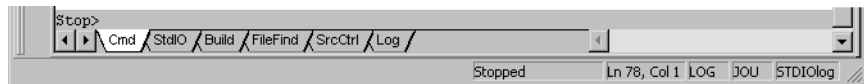


Figure 4-2 Output files recording

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, select **Tools** → **Logs and Journal** → **Close Logs/Journals....** This displays a list selection box, shown in Figure 4-3 on page 4-15, where you can specify which file, or files, to close.

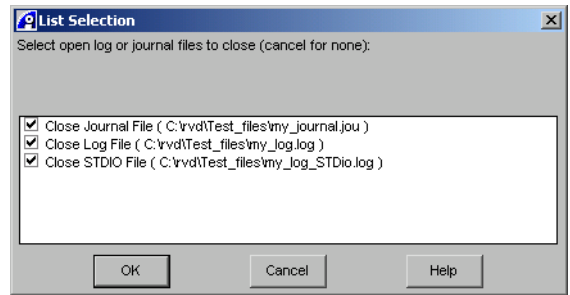


Figure 4-3 Close log and journal files selection box

Each entry has an associated check box that is ticked by default. Select a check box to unselect a file. The list selection box contains:

- OK** Click this button to close selected files and then close the selection box.
- Cancel** Click this button to leave all files open and then close the selection box. Using **Cancel** ignores the status of any check boxes in the list.
- Help** Click this button to display the online help for this selection box.

Use the File Editor pane, or a text editor of your choosing, to view the contents of your log and journal files. You can then edit the commands shown in a log file to create an include file for use as a command file or as a macro.

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 MS-DOS, or from a Command Prompt window, or create a desktop shortcut, for example:

```
rvdebug.exe -stdiolog "C:\rwd\test_files\STDIO_tst_file.log"
```

```
rvdebug.exe -jou "C:\rwd\test_files\Jou_tst_file.jou"
```

```
rvdebug.exe -log "C:\rwd\test_files\Log_tst_file.log"
```

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.

When RealView Debugger starts to write to the log file, it records the filename as the first entry, for example:

```
;;;LOG FILE: C:\rwd\test_files\my_log.log
```

4.5.2 Using include files

RealView Debugger enables you to use include files to enter commands and so carry out debugging tasks without user intervention. The commands are actioned as though they are being entered from the keyboard.

The following sections describe how to manage your debugging session using include files that are created using the logging facility, or scripts you create yourself:

- *Output buffering*
- *Including files.*

Output buffering

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.

Including files

Use a text editor to create a file of commands that can then be submitted to RealView Debugger to control a debugging session. To use an include file:

1. Select **Tools** → **Include Commands from File...**, from the Code window, to display the Select File to Include Commands from dialog box.
2. Locate the file where your debugger commands are stored. By default, RealView Debugger looks for a .inc or a .log file.
3. Click **Open** to load the file and execute the commands stored there.

You can also start RealView Debugger with an include file, for example:

```
rvdebug.exe -inc "C:\rvd\test_files\my_cmds_file.inc"
```


4.6 Searching for source files

By default RealView Debugger searches for application source file paths according to information contained in the image. If no paths are provided in the image, RealView Debugger uses the list of directories specified in the project settings. If this fails, or where no search path has been specified, RealView Debugger looks in the current working directory for the source file or files.

If you have loaded an image to a target, or where you have opened a user-defined project, you can specify search paths using the Project Properties window. Any search that you define is then saved with the project settings and additionally used to locate source files.

This section includes:

- *Example of locating source files*
- *Autoconfiguring search rules* on page 4-18
- *Manually configuring search rules* on page 4-19
- *Source path mappings* on page 4-20.

4.6.1 Example of locating source files

In the following example, the image has been built in one directory and then moved to another location for debugging. This means that RealView Debugger cannot locate source files directly from information contained in the image:

1. Ensure that the Project Properties window is not open. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open.
2. Close any files that are open in the File Editor so that they are not found, and included, in the search mechanism.
3. Connect to your target and load an image, for example `dhry_thr_demo.axf`.
RealView Debugger creates an in-memory auto-project to use for this session, that is `dhry_thr_demo.axf.apr`.
4. RealView Debugger cannot locate a source file and so displays the source search prompt, shown in Figure 4-4 on page 4-18.

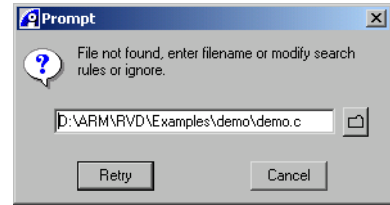


Figure 4-4 Source search prompt

If you close the prompt, for example by clicking **Cancel**, RealView Debugger warns you that it cannot locate the source file for the new image. The project properties are unchanged.

You can change the search rules that RealView Debugger uses as described in the following sections:

- *Autoconfiguring search rules*
- *Manually configuring search rules* on page 4-19.

4.6.2 Autoconfiguring search rules

Use the source search prompt, shown in Figure 4-4, to autoconfigure the search rules:

1. Either:
 - Edit the pathname shown in the prompt.
 - Click the directory button and use **<Select file...>** to locate the required file.
2. Click **Retry**.

If RealView Debugger fails to locate the file, the source search prompt remains for you to try again. If RealView Debugger succeeds, the source search prompt closes automatically.

———— Note ————

If the required file is already listed when you click the directory button, select it to update the project properties. The source search prompt then closes automatically.

Pathnames that you add this way automatically update the project properties for the active project. Where your project is an in-memory auto-project, RealView Debugger also saves the project settings file, for example `dhry_thr_demo.axf.apr`, in the same location as the original image.

Note

Absolute pathnames are used to define the new search rules when you configure them in this way. However, if you modify the source search path yourself, pathnames become relative (see *Manually configuring search rules* for details).

4.6.3 Manually configuring search rules

If you have an open project, you can specify search paths using the Project Properties window. Any search that you define is then saved with the project settings and additionally used to locate source files.

To configure the search rules manually:

1. Select **Project** → **Project Properties...** to display the Project Properties window, shown in Figure 4-5.

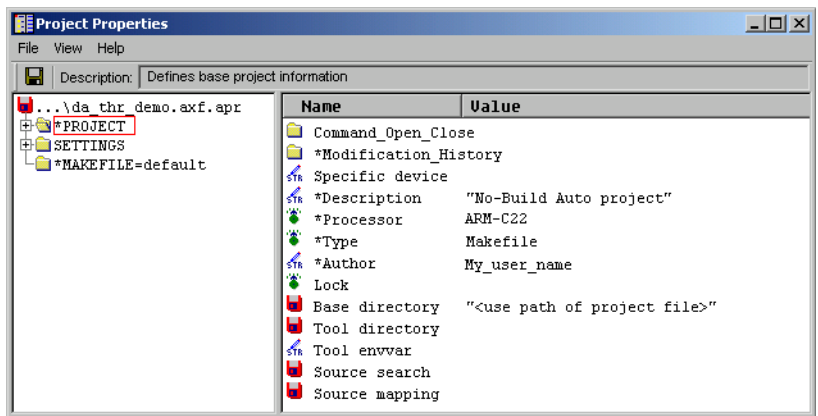


Figure 4-5 Search rules in the Project Properties window

For more details on other entries in this window see *Source path mappings* on page 4-20.

2. Right-click on Source_search and use **Edit as Directory Name...** to specify the search path to locate the missing source file.
3. Click **File** → **Save and Close** to confirm your choice and close the Project Properties window.

Do not leave the Project Properties window open when you are working with the debugger. When searching for source files, RealView Debugger updates the project properties as necessary. This fails if the window is already open.

Note

Relative pathnames are used to define the new search rules when you configure them in this way.

4.6.4 Source path mappings

When you use the directory button to locate the source file, using the prompt shown in Figure 4-4 on page 4-18, RealView Debugger creates a mapping between the original file and the new location. This mapping is then applied to subsequent file searches so that the debugger can locate files automatically that have the same mapping.

Pathname remappings are stored in the project settings so that they can be used in the next session.

Note

Where your project is an in-memory auto-project, RealView Debugger automatically saves the auto-project if you set up a new pathname mapping.

See the chapter that describes customizing projects in *RealView Developer Kit v2.2 Project Management User Guide* for full details on working with these entries.

Chapter 5

Working with 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 the following sections:

- *Breakpoints in RealView Debugger* on page 5-2
- *Setting default breakpoints* on page 5-14
- *Generic breakpoint operations* on page 5-19
- *Setting unconditional breakpoints explicitly* on page 5-20
- *Setting hardware breakpoints explicitly* on page 5-22
- *Specifying processor exceptions (global breakpoints)* on page 5-31
- *Setting conditional breakpoints* on page 5-32
- *Using the Set Address/Data Breakpoint dialog box* on page 5-39
- *Attaching macros to breakpoints* on page 5-52
- *Controlling the behavior of breakpoints* on page 5-55
- *Using the Break/Tracepoints pane* on page 5-59
- *Disabling and clearing breakpoints* on page 5-67
- *Setting breakpoints from saved lists* on page 5-69.

5.1 Breakpoints in RealView Debugger

Breakpoints are specified locations where execution must stop. The breakpoint can be triggered by:

- execution reaching the specified address
- data reads or writes at a specified address or address range
- breakpoint qualifiers passing specified test criteria
- data values at the specified location, in the current context, becoming equal to a particular value or range.

When a breakpoint triggers, RealView Debugger can carry out higher level requests. For example you can:

- attach macros to breakpoints
- output messages
- update windows or files
- change the behavior of your application program.

You can also continue execution of your application program after RealView Debugger completes the specified operations.

This section describes:

- *Breakpoint types*
- *Qualifying breakpoint line number references with module names* on page 5-6
- *Specifying address ranges* on page 5-7
- *Specifying the entry point to a function* on page 5-8
- *Breakpoints and memory map locations* on page 5-9
- *Viewing breakpoints in your code view* on page 5-10
- *Halted System Debug and Running System Debug* on page 5-12
- *Using hardware breakpoints* on page 5-12
- *Breakpoints and image restarts* on page 5-13

5.1.1 Breakpoint types

RealView Debugger enables you to use different types of breakpoint when you are debugging your image, described in:

- *Software and hardware breakpoints* on page 5-3
- *Unconditional and conditional breakpoints* on page 5-4
- *Default breakpoints in the GUI* on page 5-5
- *Recording the triggering of a breakpoint* on page 5-6.

The type of breakpoint you can set depends on the:

- memory map, if enabled
- qualifiers assigned to the breakpoint
- hardware support provided by your target processor
- target vehicle used to maintain the connection.

RealView Debugger uses breakpoints to temporarily halt execution as you step through your application. For more details about stepping through code, see *Using execution controls* on page 4-7.

Software and hardware breakpoints

RealView Debugger enables you to set software or hardware breakpoints, depending on the target and the chosen memory location (see *Breakpoints and memory map locations* on page 5-9):

Software These breakpoints are controlled by RealView Debugger directly. When you set a software breakpoint RealView Debugger modifies the executable instructions held in RAM. Therefore, you can only set software breakpoints for code stored in RAM.

You can set as many software breakpoints as you require.

———— Note ————

If you enable a Memory Management Unit (MMU) that is configured to set the region of memory containing a breakpoint to read-only, then the breakpoint cannot be removed. In this case, you must make sure you clear the breakpoints before enabling the MMU.

Hardware Hardware breakpoints use advanced hardware support on your target processor, or as implemented by your simulator software. The complexity and number of breakpoints you can set depends on:

- hardware support provided by your target processor
- target vehicle used to maintain the connection.

Hardware breakpoints are controlled by the EmbeddedICE® module of your target, which can be physical hardware or a software model.

———— Note ————

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

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

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

Check your hardware characteristics, and your vendor-supplied documentation, to determine the level of support available for hardware breakpoints. Also, see *Viewing your hardware breakpoint support* on page 5-12.

Unconditional and conditional breakpoints

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

Unconditional

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

You can set an unconditional breakpoint that has no condition qualifiers and no actions assigned, and without explicitly choosing whether it is a software or hardware breakpoint. This is called a *default breakpoint*, and either RealView Debugger or your target hardware determines what kind of breakpoint is set. For a default breakpoint, the program always stops if execution reaches a specified location (see *Setting default breakpoints* on page 5-14).

You can assign one or more actions to an unconditional breakpoint. When the breakpoint is triggered, the specified action is performed (see *Setting unconditional breakpoints explicitly* on page 5-20).

Conditional A breakpoint is conditional when a condition qualifier is assigned to the breakpoint. The conditions that must be satisfied can be defined based on data values, the result of a macro, or on counters. See *Controlling the behavior of breakpoints* on page 5-55 for details on how the order of conditional qualifiers affects the behavior of a breakpoint.

The program stops if execution reaches a specified location and one or more of the predefined conditions are met (see *Setting conditional breakpoints* on page 5-32).

Note

Although hardware breakpoints can have conditions controlled in hardware, they are not conditional in RealView Debugger. Only breakpoints that have condition qualifiers assigned are identified as conditional by RealView Debugger.

Default breakpoints in the GUI

A *default breakpoint* is set by RealView Debugger when you do not explicitly choose a specific type of breakpoint. For example, a default breakpoint is set when you double-click in the margin of the **Dsm** tab or a source code tab of the File Editor pane.

A default breakpoint does not have any condition qualifiers or actions, and so is always an unconditional breakpoint. However, the chosen memory location where you set the breakpoint determines whether it is a software or hardware breakpoint (see *Breakpoints and memory map locations* on page 5-9).

The command used by RealView Debugger to set a default breakpoint depends on the type:

- To set a default software breakpoint, RealView Debugger uses the BREAKINSTRUCTION command. For example, to set a default breakpoint at the RAM address 0x00008008, RealView Debugger issues the command:
`binstr (I A)0x00008008`
- To set a default hardware breakpoint, RealView Debugger uses the BREAKEXECUTION command. For example, if you set a default breakpoint at the Integrator™/AP Flash address 0x24000008, RealView Debugger issues the command:
`bexec (I A)0x24000008`

Note

In some instances, RealView Debugger might attempt to set a software breakpoint, but your target hardware might set a hardware breakpoint instead. In this case, the breakpoint information in the Break/Tracepoints pane might show Exec in the breakpoint title, but the bi command (BREAKINSTRUCTION) for the command used to set the breakpoint.

If you attempt to set a default breakpoint in a data or literals area of code within the **Dsm** tab, the breakpoint is set and RealView Debugger issues the warning:

Warning: Breakpoint being set in data/literals of code.

For more details about these commands, see the *RealView Developer Kit v2.2 Command Line Reference*.

For detailed instructions on setting default breakpoints, see *Setting default breakpoints* on page 5-14.

Note

Default breakpoints are not added to the breakpoint history (see *Setting breakpoints from saved lists* on page 5-69).

Recording the triggering of a breakpoint

When a breakpoint triggers and program execution stops, RealView Debugger records this by displaying messages that identify the type and location of the breakpoint, for example:

```
Stopped at 0x00008498 due to SW Instruction Breakpoint
Stopped at 0x00008498: DHRY_1\main Line 153
```

If you specify that execution is to continue after the breakpoint is triggered, then these messages are not displayed.

If you want program execution to continue when a breakpoint is triggered, and still be informed when this occurs, then:

1. Assign a macro to the breakpoint that outputs appropriate messages and returns a value of zero (True).
2. Assign the continue command qualifier to the breakpoint.

See *Controlling the behavior of breakpoints* on page 5-55 for more details.

5.1.2 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 can specify only a line number in the file. For example, using the BREAKINSTRUCTION command you can set a breakpoint at column 2 of line 92 in the dhry_1.c source file with:

```
breakinstruction #92:2
```

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.

Module naming conventions

For sources with the .c or .cpp extension, the module name is the source filename without the extension. If the extension is not .c or .cpp, 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 indicating a structure reference.

For example, to set a breakpoint at line 48 in the file dhry_2.c for the image dhrystone.axf, enter:

```
breakinstruction \DHRY_2\#48:1
```

5.1.3 Specifying address ranges

Hardware breakpoints enable you to specify a range of addresses. 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..0x20000

start_addr..+length

Start address and length of the address region, for example:

0x10000..+0x10000

In both cases, the start and end values are inclusive.

You can also use symbol names, such as function names, as the start address. For example:

main..+1000Arr_2_Glob..+64

5.1.4 Specifying the entry point to a function

You can set a breakpoint at the entry point of a function with the `@entry` symbol. This enables you to set a breakpoint on a function without having to open the source file containing that function, and without having to scroll down the Dsm tab to find it.

To set a breakpoint on the entry to a function, specify *function\@entry*. 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 auto local that is initialized in that 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 triggers only when the argument value has a specific value on entry to the function:

```
bi,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` example image:

- specifying `main` sets a breakpoint at `0x000083D0`, the first address in the function (line 78 in `dhry_1.c`)
 - specifying `main\@entry` sets a breakpoint at `0x000083D8`, which is the first line of executable code (line 91 in `dhry_1.c`).
-

5.1.5 Breakpoints and memory map locations

With memory mapping enabled, RealView Debugger sets a breakpoint based on the access rule for the memory at the chosen location. Table 5-1 shows the different types of location that you can define in a memory map with RealView Debugger, and the type of breakpoint that RealView Debugger sets by default.

Table 5-1 Breakpoint types for different memory map locations

Location	Breakpoint type set
RAM	Software
ROM	Hardware
Flash	Hardware
Write-only Memory (WOM)	RealView Debugger prompts you
No Memory (NOM)	Hardware
Auto	Hardware

If you have to, you can set a hardware breakpoint in RAM. To do this, you must use one of the methods described in *Setting hardware breakpoints explicitly* on page 5-22.

RealView Debugger cannot determine the type of default breakpoint to set in WOM memory, and displays the Set Address/Data Breakpoint dialog box (see *Displaying the Set Address/Data Breakpoint dialog box* on page 5-39). This dialog box is also displayed if an error occurs when you attempt to set a breakpoint.

If you attempt to set a software breakpoint at an address in Flash, ROM, NOM, or Auto memory, a hardware breakpoint is set instead. If the target vehicle informs RealView Debugger that a hardware breakpoint is being used, RealView Debugger warns you that this has happened:

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

For more details about memory mapping, see Chapter 6 *Memory Mapping*.

5.1.6 Viewing breakpoints in your code view

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 (see *Breakpoint types* on page 5-2) 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 set a conditional breakpoint, and that the breakpoint is enabled. The conditions are software conditions that are controlled by RealView Debugger. Although hardware breakpoints can have conditions that are controlled in hardware, only when the software conditions are assigned are these breakpoints conditional in RealView Debugger.
- White** A white icon shows that you have disabled an existing breakpoint. When you re-enable the breakpoint, the icon changes back to the color it had before the breakpoint was disabled.

This color coding is also reflected in the Break/Tracepoints pane.

Note

Hardware access, read, and write breakpoints are not shown in the File Editor pane, because there is no specific line or context to display them.

If you try to set a breakpoint on a non-executable line, RealView Debugger looks for the first executable line immediately following and places the breakpoint there. If the lines preceding the breakpointed instruction are comments, declarations, or other non-executable code, they are marked with black, downward pointing arrows. Lines marked in this way are regarded as part of the breakpoint.

Multiple breakpoints units on a single line

If you have set multiple breakpoint units on a source line containing multiple statements or on inline functions, then disabling the first breakpoint unit changes the marker disc, shown in Figure 5-1 on page 5-11. If you disable the second breakpoint unit on the line, the marker remains.

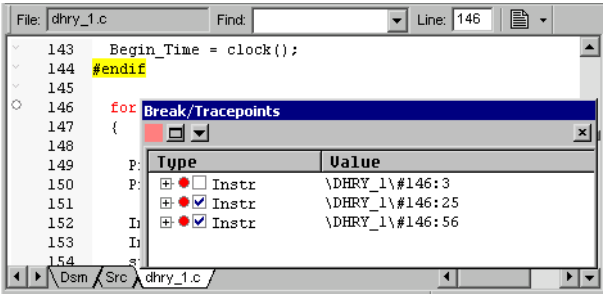


Figure 5-1 Working with multiple breakpoint units

See *Using the Break/Tracepoints pane* on page 5-59 for details on viewing breakpoints in the Break/Tracepoints pane.

You cannot place two breakpoints of the same type at the same line or on lines marked by downward pointing arrows. For example, you cannot set two software breakpoints on the same line.

Clearing breakpoints quickly

With a breakpoint visible in your current code view, you can clear it quickly by double-clicking on the marker icon in the margin. This removes the breakpoint you set.




———— **Note** —————

Where you have set multiple breakpoint units, clearing a breakpoint this way removes only the first breakpoint unit.

Other types of breakpoints

If you have set thread breakpoints, these are shown as different types of marker using icons as shown in Table 5-2. See *Halted System Debug and Running System Debug* on page 5-12 for details of using these features.

Table 5-2 Other breakpoint/tracepoint icons

Icon	Breakpoint/Tracepoint type
	HSD breakpoint
	RSD Thread breakpoint
	RSD System breakpoint

5.1.7 Halted System Debug and Running System Debug

RealView Debugger supports different debugging modes:

Halted System Debug

Halted System Debug (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. This debugging mode places no demands on the application running on the target.

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

5.1.8 Using hardware breakpoints

Setting a software breakpoint requires that the debugger changes executable instructions, so this is only possible for code stored in RAM. Where instructions are in Flash or ROM, you must set hardware breakpoints.

The number of hardware breakpoints available depends on your debug target. If RealView Debugger cannot set breakpoints, a warning message is displayed and rapid instruction step is used for high-level stepping. Again, a warning message is displayed to explain the type of step being used.

To use hardware breakpoints, your debug target must include support for such breakpoints. Even where this support is available, your target might be limited in the number it can support at one time. RealView Debugger menu options related to hardware breakpoints are grayed out if your target cannot support them or if no more are available.

Viewing your hardware breakpoint support

To see your hardware support for breakpoints, select the following option from the Code window main menu:

Debug → Breakpoints → Hardware → Show Break Capabilities of HW...

This displays an information box describing the support available for your target processor, shown in Figure 5-2 on page 5-13.

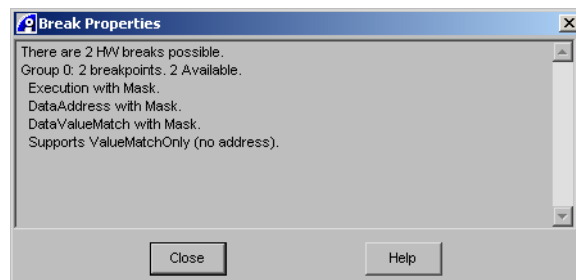


Figure 5-2 Example hardware break characteristics

What is shown in this details display depends on the target processor and the target vehicle used to make the connection. Figure 5-2 shows the details for an ARM946E-S™ using RVI-ME.

———— **Note** ————

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

5.1.9 Breakpoints and image restarts

When your image stops executing, either by manual intervention or by terminating naturally, the PC no longer points to the image entry point. If you want to restart the image again from the image entry point, you can use one of the following methods:

- unload the image, then load it again (two-step operation)
- reload the image (single-step operation)
- set the PC to the image entry point.

However, if you have set any breakpoints, the first method causes them to be cleared. If you want to preserve any existing breakpoints, then use either of the other methods. See Chapter 4 *Controlling Execution* for more details.

5.1.10 Using breakpoints with RealView ICE and RVI-ME

If you are using RealView ICE or RVI-ME, RealView Debugger sets breakpoints differently. For full details on debugging with RealView ICE and RVI-ME, see the following documentation for information:

- *RealView ICE and RealView Trace User Guide*
- *RealView ICE Micro Edition v1.1 User Guide.*

5.2 Setting default breakpoints

You can set a default breakpoint directly from the Code window (see *Default breakpoints in the GUI* on page 5-5). The methods you can use to do this depend on your current code view. A default breakpoint is a useful way to set a quick test point during a debugging session.

The type of default breakpoint set, either software or hardware, is determined by the chosen memory location (see *Breakpoints and memory map locations* on page 5-9).

Note

Default breakpoints are not added to the breakpoint history (see *Setting breakpoints from saved lists* on page 5-69).

This section describes:

- *Default breakpoints in source-level view* on page 5-15
- *Default breakpoints in disassembly-level view* on page 5-16
- *Setting breakpoints on branch instructions* on page 5-17
- *Setting default breakpoints in other ways* on page 5-18.

In these examples, **Text Coloring** is enabled by default. To turn on line numbering, select the following option from the Code window main menu:

Edit → Advanced → Show Line Numbers

Note

This menu option is effective only for the current debugging session. If you want to show line numbers by default when you start RealView Debugger, change the `Line_number` setting in the workspace settings (see *Edit* on page A-12).

5.2.1 Default breakpoints in source-level view

Double-click in the margin, that is the gray area to the left of a line, to set a default breakpoint quickly, shown in Figure 5-3. You can also double-click on the line number if this is visible.

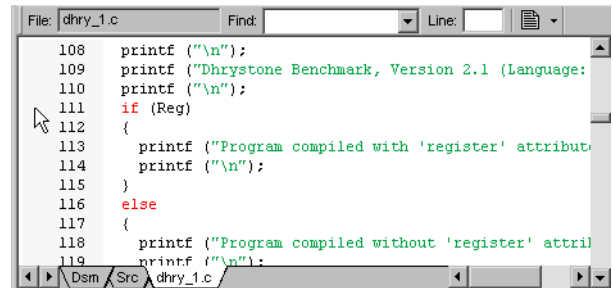


Figure 5-3 Setting an unconditional breakpoint on a line

This example sets a default software breakpoint marked by a red disc in the margin.

Setting a breakpoint on a multi-statement line

If you want to set a default breakpoint on a multi-statement line, for example on a for... loop, right-click on the statement to display either the **Variable** or **Symbol** context menu, and select **Set Break on Statement**.

Setting a breakpoint on a symbol

If you want to set a breakpoint on a symbol, right-click on the symbol (for example, a function name) to display the **Symbol** context menu, and select **Set Break On**. You can view the currently defined symbols with the PRINTSYMBOL command, described in the *RealView Developer Kit v2.2 Command Line Reference*.

Note

The options you can use on the **Variable** context menu and **Symbol** context menu depend on which part of the statement you right-click.

Setting a breakpoint in the body of a function

You can also use this option to set a breakpoint in the body of a function by right-clicking on the call to the function. For example:

1. Locate line 156 in `dhry_1.c`.
2. Right-click on the function name `Func_2`.
3. Select **Set Break On** from the **Symbol** context menu, to set the breakpoint.
4. Right-click on the function name `Func_2` again.
5. Select **Scope To** from the **Symbol** context menu.

RealView Debugger opens the source `dhry_2` at the function `Func_2`, and shows the breakpoint set at line 143.

Effects of setting a breakpoint

Setting a breakpoint updates the Break/Tracepoints pane, if it is visible, and the Output pane shows the breakpoint command, for example:

```
bi \DHR_1\ #146:3
```

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 where the code is in RAM.

If your code is in ROM or Flash, RealView Debugger sets a hardware breakpoint where one is available. This is set using a `BREAKEXECUTION` command. An error message is displayed if no such breakpoint is available.

See *Using the Break/Tracepoints pane* on page 5-59 for details on viewing breakpoints in the Break/Tracepoints pane.

————— Note —————

For details of the `BREAKINSTRUCTION` and `BREAKEXECUTION` commands see *RealView Developer Kit v2.2 Command Line Reference*.

5.2.2 Default breakpoints in disassembly-level view

To set a default breakpoint quickly, double-click on the margin to the left of the required instruction in the **Dsm** tab.

5.2.3 Setting breakpoints on branch instructions

To set a default breakpoint on the destination of a branch instruction:

1. Locate the required branch instruction in the **Dsm** tab.
2. Right-click on the destination address specified in the branch instruction to display the **Instruction** context menu.

———— Note ————

If you right-click on the background, the **File Editor** context menu is displayed. You must right-click on the text of the instruction, or its address, to display the **Instruction** context menu. Also, the options you can use on the **Instruction** context menu depend on which part of the instruction you right-click.

3. Select **Set Instr Break** from the **Instruction** context menu.

The breakpoint is set on the instruction at the specified branch address, shown in Figure 5-4.

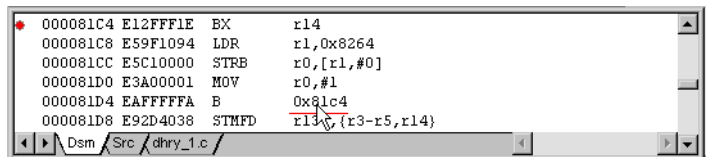


Figure 5-4 Setting an unconditional breakpoint on an instruction

4. The Break/Tracepoints pane is updated with the new breakpoint (if visible) and the Output pane shows the breakpoint command:

```
bi 0x81c4
```

———— Note ————

If the location where the breakpoint is set is not currently visible, right-click on the branch address and select **Scope To** from the **Symbol** context menu to view the location.

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 *Using the Break/Tracepoints pane* on page 5-59 for details on viewing breakpoints in the Break/Tracepoints pane.

5.2.4 Setting default breakpoints in other ways

RealView Debugger includes other ways to set default breakpoints and manipulate existing ones:

- Use one of the generic breakpoint operations described in *Generic breakpoint operations* on page 5-19.
- Use drag-and-drop to create a breakpoint in the Break/Tracepoints pane based on a program item. For example, highlight a source code function in the File Editor pane and then drag it (using your mouse) and drop it into the Break/Tracepoints pane (see *Using the Break/Tracepoints pane* on page 5-59 for details).
- Right-click in the left margin of the File Editor pane, and select **Set Break** from the context menu.
- Right-click on a function in the **Functions** tab of the Data Navigator pane, and select **Break** from the context menu. See *Using the Data Navigator pane* on page 9-5 for more details on working with the Data Navigator pane.

5.3 Generic breakpoint operations

The **Debug** → **Breakpoints** menu enables you to complete certain breakpoint operations that are common to all breakpoints:

Toggle Break at Cursor

Toggles a breakpoint at the address defined by the position of the cursor in your code view:

- If no breakpoint exists at the address, a new default breakpoint is created (see *Default breakpoints in the GUI* on page 5-5).
- If a breakpoint already exists at the address, it is cleared.

Enable/Disable Break at Cursor

Enables or disables a breakpoint at the address defined by the position of the cursor in your code view:

- If an enabled breakpoint already exists at the address, it is disabled.
- If a disabled breakpoint already exists at the address, it is enabled.
- If no breakpoint exists at the address, a new default breakpoint is created (see *Default breakpoints in the GUI* on page 5-5).

Clear All Break/Tracepoints

Clears all breakpoints, and tracepoints, that you have set on the current target.

See *Disabling and clearing breakpoints* on page 5-67 for more details of disabling and clearing breakpoints.

5.4 Setting unconditional breakpoints explicitly

You can set an unconditional breakpoint explicitly using one of the following methods:

- use the Simple Break if X dialog box (see *Using the Simple Break if X dialog box*)
- use the Set Address/Data Breakpoint dialog box without assigning any qualifiers (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).

5.4.1 Using the Simple Break if X dialog box

The Simple Break if X dialog box enables you to set an unconditional software or hardware breakpoint, depending on the memory location.

To display this dialog box, shown in Figure 5-5, use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Conditional → Break if X...
- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set BreakIf...** from the context menu. Then select **Break if X...** from the List Selection dialog box, and click **OK**.
- Using the Break/Tracepoints pane, either:
 - select **Set BreakIf...** from the **Pane** menu
 - right-click on the pane background, and select **Set BreakIf...** from the context menu.

Then select **Break if X...** from the List Selection dialog box, and click **OK**.

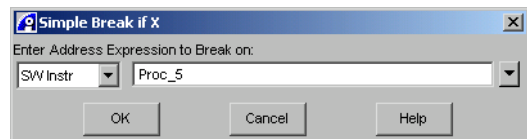


Figure 5-5 Simple Break if X dialog box

The Breakpoint Type controls the type of memory, program, or data, and the type of access that stops execution. In this case, this shows **SW Instr** as the given type. This field is set to read-only where this is the only type of breakpoint supported by your debug target.

If your target supports hardware breakpoints, click on the drop-down arrow to display a list of the available types.

Specify the location to break on. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 5-6)
- a specific address or address range (see *Specifying address ranges* on page 5-7)
- a function entry (see *Specifying the entry point to a function* on page 5-8).

You can enter a location directly in the field, or click the drop-down arrow to the right of the field to choose a location from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

5.4.2 Setting unconditional breakpoints in other ways

RealView Debugger includes other ways for you to explicitly set new unconditional breakpoints and manipulate existing ones:

- Set a breakpoint on a memory location in the Memory pane. The type of breakpoint offered depends on the type of memory at the chosen location, for example RAM, ROM, or Flash. See *Operating on memory contents* on page 7-17 for details.
- Set a hardware breakpoint at an address of a symbol using the **Set Break At...** option on the Stack pane context menu, but do not assign a qualifier to the breakpoint. This displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).

However, this is not recommended if execution runs past the end of a function return call because as soon as you exit the function the stack value is no longer meaningful.

Use the Watch pane to track a specific symbol continuously because this recomputes the stack location dynamically and so tracks each invocation of the function.

You can use local variables within the conditional part of a breakpoint because the stack value is computed correctly each time the breakpoint condition is evaluated.

- You can use the Data Navigator pane to set an unconditional breakpoint on a chosen function or variable, defined as a location in the image. See *Using the Data Navigator pane* on page 9-5 for details.

5.5 Setting hardware breakpoints explicitly

Use hardware breakpoints to set execution or data breaks, or where your code is stored in ROM or Flash (see *Breakpoints and memory map locations* on page 5-9). The facilities available depend on the current debug target, that is both the target processor and the target vehicle.

To set hardware breakpoints, you must be connected to a debug target that supports these features, such as an ARM processor with EmbeddedICE logic (for example, an ARM966E-S).

Menu options related to hardware breakpoints are grayed out if your target cannot support them. Similarly, RealView Debugger displays a message if you select an option from a drop-down list box that is not supported by your debug target. Check your hardware characteristics (see *Viewing your hardware breakpoint support* on page 5-12), and your vendor-supplied documentation, to determine the level of support for hardware breakpoints.

This section includes:

- *Using the HW Break if in Range dialog box*
- *Using the HW While in function/range, Break if X dialog box* on page 5-24
- *Using the HW Break if X, then if Y dialog box* on page 5-25
- *Using the HW Break on Data Value match dialog box* on page 5-27
- *Chaining breakpoints* on page 5-29.

———— Note ————

The methods described in this section create unconditional hardware breakpoints. To set conditional hardware breakpoints, see *Setting conditional hardware breakpoints* on page 5-36.

5.5.1 Using the HW Break if in Range dialog box

The HW Break if in Range dialog box enables you to set, or modify, a hardware breakpoint at the specified location. A single breakpoint instance is created. The breakpoint is triggered if the PC is within the given address range and, optionally, data matches a specified value.

Displaying the dialog box

To display the HW Break if in Range dialog box, shown in Figure 5-6 on page 5-23, select the following option from the Code window main menu:

Debug → Breakpoints → Hardware → HW Break if in Range...

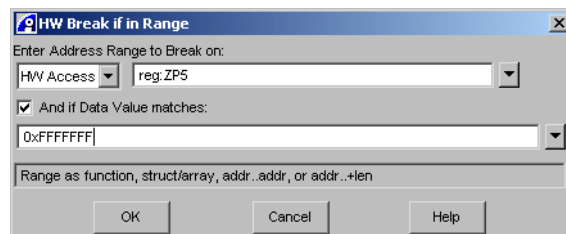


Figure 5-6 HW Break if in Range dialog box

HW Break if in Range dialog box interface components

The HW Break if in Range dialog box (see Figure 5-6) contains the following fields:

Enter Address Range to Break on

Choose the type of hardware breakpoint that you want to set. The options are:

- **HW Access**
- **HW Read**
- **HW Write**
- **HW Instr.**

Specify the location to break on. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 5-6)
- a specific address or address range (see *Specifying address ranges* on page 5-7)
- a function entry (see *Specifying the entry point to a function* on page 5-8).

And if Data Value matches

Select this check box if you also want to test a data value to trigger the breakpoint. Enter the data value to test in the data field.

You can enter values directly in these fields, or click the drop-down arrow to the right of these fields to choose from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

5.5.2 Using the HW While in function/range, Break if X dialog box

The HW While in function/range, Break if X dialog box enables you to specify a hardware breakpoint that uses two chained breakpoint units (see *Chaining breakpoints* on page 5-29 for more information).

Displaying the dialog box

To display the HW While in function/range, Break if X dialog box, shown in Figure 5-7, select the following option from the Code window main menu:

Debug → Breakpoints → Hardware → HW While in function/range, Break if X...

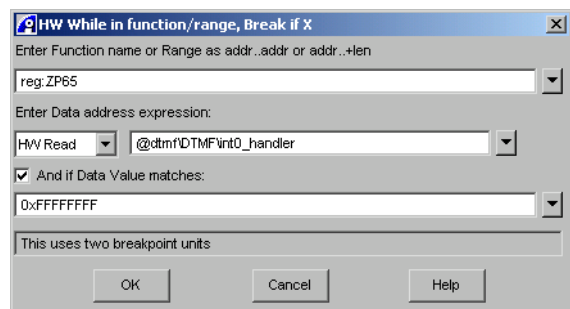


Figure 5-7 HW While in func/range, Break if X dialog box

HW While in function/range, Break if X interface components

The HW While in function/range, Break if X dialog box (see Figure 5-7) contains the following fields:

Enter Function name or Range

Specify the function or location to test for the first breakpoint unit. This can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 5-6)
- a specific address or address range (see *Specifying address ranges* on page 5-7)
- a function entry (see *Specifying the entry point to a function* on page 5-8).

The breakpoint unit triggers if the PC equals the corresponding address, or falls within the specified address range.

Enter Data address expression

Choose the type of hardware breakpoint that you want to set. The options are:

- **HW Access**
- **HW Read**
- **HW Write.**

Enter the data address expression that must be accessed to trigger the breakpoint, for example:

```
@dtmf\DTMF\int0_handler
```

And if Data Value matches

Select this check box if you want to test a data value to trigger the breakpoint. Enter the data value to test in the data field.

You can enter values directly in these fields, or click the drop-down arrow to the right of these fields to choose from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

5.5.3 Using the HW Break if X, then if Y dialog box

The HW Break if X, then if Y dialog box enables you to specify a hardware breakpoint that uses two chained breakpoint units (see *Chaining breakpoints* on page 5-29 for more information). The breakpoint is set at the specified memory location or address range (see *Specifying address ranges* on page 5-7) depending on the values read.

Displaying the HW Break if X, then if Y dialog box

To display the HW Break if X, then if Y dialog box, shown in Figure 5-8, select the following option from the Code window main menu:

Debug → Breakpoints → Hardware → HW Break if X, then if Y...

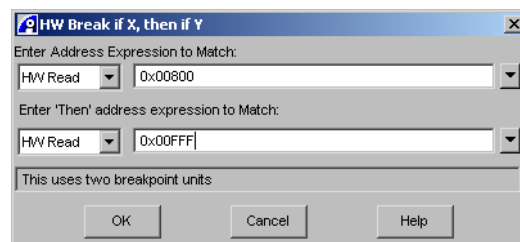


Figure 5-8 HW break if X, then if Y dialog box

HW Break if X, then if Y dialog box interface components

The HW Break if X, then if Y dialog box (see Figure 5-8 on page 5-25) contains the following fields:

Enter Address Expression to Match

Choose the type of hardware breakpoint that you want to set for the first breakpoint unit. The options are:

- **HW Access**
- **HW Read**
- **HW Write**
- **HW Instr.**

Specify the location to test for the first breakpoint unit (X). The breakpoint unit triggers if the PC equals the corresponding address, or falls within the specified address range.

Enter 'Then' address expression to Match

Choose the type of hardware breakpoint that you want to set for the second breakpoint unit. The options are:

- **HW Access**
- **HW Read**
- **HW Write**
- **HW Instr.**

Specify the location to test for the second breakpoint unit (Y). The breakpoint unit triggers if the PC equals the corresponding address, or falls within the specified address range.

The location where the breakpoint is to be set can be:

- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 5-6)
- a specific address or address range (see *Specifying address ranges* on page 5-7)
- a function entry (see *Specifying the entry point to a function* on page 5-8).

You can enter the location directly into these fields, or click the drop-down arrow to the right of these fields to choose a location from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

5.5.4 Using the HW Break on Data Value match dialog box

The HW Break on Data Value match dialog box enables you to specify the range of data values to test for the breakpoint. For example the low value, and the high value, or you can use a mask. The breakpoint triggers if the PC falls within the specified range. A single breakpoint instance is created with this dialog box.

Displaying the HW Break on Data Value match dialog box

To display the HW Break on Data Value match dialog box, shown in Figure 5-9, use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Hardware → HW Break on Data Value match...
- Using the Break/Tracepoints pane, either:
 - select **Set BreakIf...** from the **Pane** menu
 - right-click on the pane background, and select **Set BreakIf...** from the context menu.

Then select **HW Break on Data Value match...** from the List Selection dialog box, and click **OK**.

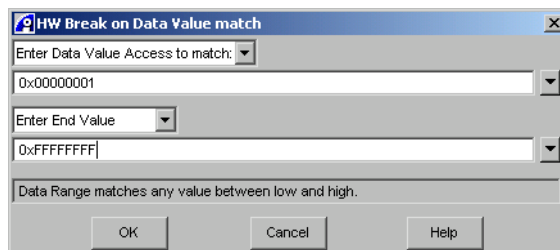


Figure 5-9 HW Break on Data Value match dialog box

HW Break on Data Value match dialog box interface components

The HW Break on Data Value match dialog box (see Figure 5-9) contains the following fields:

Data value to match

Choose the type of hardware breakpoint that you want to set for the match. This can be one of the following:

- **Enter Data Value Access to Match**
- **Enter Data Value Read to Match**
- **Enter Data Value Write to Match.**

Enter the data value to test against in the data field.

Data value modifier

Choose a data value modifier, if your hardware supports it. This can be one of the following:

Enter None (no range)

Test against the first data value only. This is the default.

Enter End Value

Test for values in the range from the first data value and the second data value specified, inclusive.

Enter Value Length

Test for values in the range from the first data value and 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.

You can enter values directly in these fields, or click the drop-down arrow to the right of these fields to choose from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

5.5.5 Chaining breakpoints

You can create complex conditional breakpoints by chaining individual breakpoints together. Only hardware breakpoints can be chained together. You can create chained breakpoints in the following ways:

- Using the following dialog boxes:
 - HW While in function/range, Break if X dialog box (see *Using the HW While in function/range, Break if X dialog box* on page 5-24)
 - HW Break if X, then if Y dialog box (see *Using the HW Break if X, then if Y dialog box* on page 5-25).
- Using break command qualifiers, see:
 - *Chaining breakpoints with command qualifiers*
 - *Converting existing individual breakpoints to chained breakpoints.*

Chaining breakpoints with command qualifiers

You use the `hw_and` command qualifier to create chained breakpoints. For the first breakpoint in the chain, include the `hw_and:next` or `hw_and:"then-next"` command qualifiers. For each subsequent breakpoint in the chain, include the `hw_and:"then-prev"` or `hw_and:prev` command qualifiers.

For more details on the break commands, see the *RealView Developer Kit v2.2 Command Line Reference*

Converting existing individual breakpoints to chained breakpoints

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 a chained breakpoint with any other qualifier and also change the address expression.

To create chained breakpoints based on existing breakpoints:

1. Display the Break/Tracepoints pane.
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.

For more details on the break commands, see the *RealView Developer Kit v2.2 Command Line Reference*.

5.6 Specifying processor exceptions (global breakpoints)

If supported by your debug target, RealView Debugger maintains a list of processor exceptions that automatically trigger 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.

5.6.1 Displaying the List Selection dialog box

To display the List Selection dialog box, shown in Figure 5-10, use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Processor Exceptions...
- Using the Break/Tracepoints pane, either:
 - select **Processor Exceptions...** from the **Pane** menu
 - right-click on the pane background, and select **Processor Exceptions...** from the context menu.

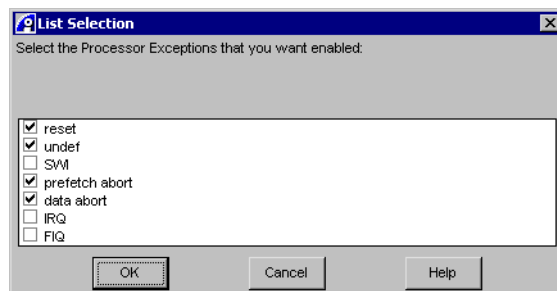


Figure 5-10 Processor Exceptions list selection box

5.6.2 List Selection dialog box interface components

The List Selection dialog box (see Figure 5-10) shows processor exceptions that stop execution. An exception is enabled when the associated check box contains a tick:

1. Click on a check box to enable, or disable, a chosen exception.
2. Click **OK** to make the chosen exceptions active.

5.7 Setting conditional breakpoints

Conditional breakpoints have condition qualifiers assigned to test for conditions that must be satisfied to trigger the breakpoint. For example, you can:

- test a variable for a given value
- execute a function a set number of times
- run a macro.

This section includes:

- *Considerations when using conditional breakpoints*
- *Using the Simple Break if X, N times dialog box on page 5-33*
- *Using the Simple Break if X, when Y is True dialog box on page 5-35*
- *Setting conditional hardware breakpoints on page 5-36*
- *Setting conditional breakpoints in other ways on page 5-37.*

5.7.1 Considerations when using conditional breakpoints

Conditional breakpoints can be very intrusive because RealView Debugger takes control when the breakpoint triggers. The specified qualifiers are checked and then, if applicable, control is returned to the application. See *Controlling the behavior of breakpoints* on page 5-55 for details on how the order of conditional qualifiers affects the behavior of a breakpoint.

When a conditional breakpoint triggers, 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 triggered. Depending on the tasks on the 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 speed of your debug target and the processor, or processors, it contains
- those instructions currently being executed on the target
- how much processing is required on the host to resolve the conditional breakpoint.

5.7.2 Using the Simple Break if X, N times dialog box

The Simple Break if X, N times dialog box, shown in Figure 5-11, enables you to set a conditional breakpoint that specifies how many times execution must arrive at the specified address before the breakpoint triggers. This dialog box provides a quick way of assigning the **SW Pass Count** qualifier to a breakpoint (see *Specifying Qualifiers* on page 5-48).

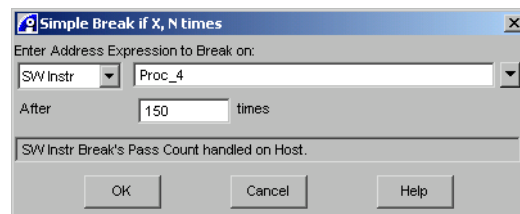


Figure 5-11 Simple Break if X, N times dialog box

Displaying the Simple Break if X, N times dialog box

To display the Simple Break if X, N times dialog box (see Figure 5-11) use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Conditional → Break if X, N times...
- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set BreakIf...** from the context menu. Then select **Break if X, N times...** from the List Selection dialog box, and click **OK**.
- Using the Break/Tracepoints pane, either:
 - select **Set BreakIf...** from the **Pane** menu
 - right-click on the pane background, and select **Set BreakIf...** from the context menu.
 Then select **Break if X, N times...** from the List Selection dialog box, and click **OK**.
- Right-click on a value shown in the Watch pane, or on a variable shown in the Call Stack pane, and select **BreakIf...** from the context menu. Then select **Break if X, N times...** from the List Selection dialog box, and click **OK**.

Simple Break if X, N times dialog box interface components

The Simple Break if X, N times dialog box (see Figure 5-11 on page 5-33) contains the following fields:

Enter Address Expression to Break on

Choose the type of breakpoint that you want to set. The options are:

- **SW Instr**
- **HW Instr**
- **HW Access**
- **HW Read**
- **HW Write.**

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 (see *Qualifying breakpoint line number references with module names* on page 5-6)
- a specific address or address range (see *Specifying address ranges* on page 5-7)
- a function entry (see *Specifying the entry point to a function* on page 5-8).

The breakpoint unit triggers 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 a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

After The number of times execution must arrive at the specified address to trigger the breakpoint, for example, when Proc_4 has been executed 150 times.

———— Note —————

If you are using a debug target that supports it, the pass count can be made in hardware.

5.7.3 Using the Simple Break if X, when Y is True dialog box

The Simple Break if X, when Y is True dialog box, shown in Figure 5-12, enables you to set a conditional breakpoint that specifies an expression (given in C format). The result must be True when execution arrives at the specified address for the breakpoint to be triggered. This dialog box provides a quick way of assigning the **When Expression True** qualifier to a breakpoint (see *Specifying Qualifiers* on page 5-48).

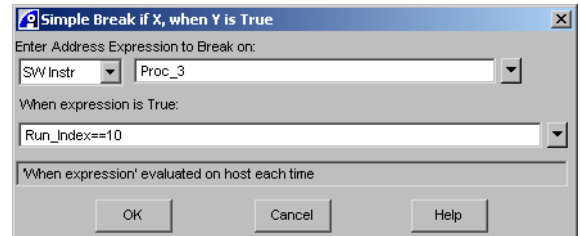


Figure 5-12 Simple Break if X, when Y is True dialog box

Displaying the Simple Break if X, when Y is True dialog box

To display the Simple Break if X, when Y is True dialog box (see Figure 5-12) use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Conditional → Break if X, when Y is True...
- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set BreakIf...** from the context menu. Then select **Break if X, when Y is True...** from the List Selection dialog box, and click **OK**.
- Using the Break/Tracepoints pane, either:
 - select **Set BreakIf...** from the **Pane** menu
 - right-click on the pane background, and select **Set BreakIf...** from the context menu.

Then select **Break if X, when Y is True...** from the List Selection dialog box, and click **OK**.

- Right-click on a value shown in the Watch pane, or on a variable shown in the Call Stack pane, and select **BreakIf...** from the context menu. Then select **Break if X, when Y is True...** from the List Selection dialog box, and click **OK**.

Simple Break if X, when Y is True dialog box interface components

The Simple Break if X, when Y is True dialog box (see Figure 5-12 on page 5-35) contains the following fields:

Enter Address Expression to Break on

Choose the type of breakpoint that you want to set. The options are:

- **SW Instr**
- **HW Instr**
- **HW Access**
- **HW Read**
- **HW Write.**

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 (see *Qualifying breakpoint line number references with module names* on page 5-6)
- a specific address or address range (see *Specifying address ranges* on page 5-7)
- a function entry (see *Specifying the entry point to a function* on page 5-8).

The breakpoint unit triggers if the PC equals the corresponding address, or falls within the specified address range. For example, Proc_3.

When expression is True

The expression to test. This must give a True or False value as the result, or example, Run_Index==10.

You can enter values directly in these fields, or you can click the drop-down arrow to the right of the fields to choose from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

5.7.4 Setting 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 condition qualifier is assigned to the breakpoint. Condition qualifiers are software conditions that are controlled by RealView Debugger.

If you have created a hardware breakpoint using one of the methods described in *Setting hardware breakpoints explicitly* on page 5-22, then you must edit the breakpoint to assign one or more condition qualifiers to it. To do this, right-click on the breakpoint entry in the Break/Tracepoints pane, and select **Edit Break/Tracepoint...** from the context menu. The Set Address/Data Breakpoint dialog box is displayed, where you can assign condition qualifiers to the breakpoint.

For details on using the Set Address/Data Breakpoint dialog box, see *Using the Set Address/Data Breakpoint dialog box* on page 5-39.

5.7.5 Setting conditional breakpoints in other ways

RealView Debugger includes other ways to set conditional breakpoints and manipulate existing ones:

- Use drag-and-drop to create a breakpoint in the Break/Tracepoints pane based on a program item, for example highlight a source code function in the File Editor pane and then drag it (using your mouse) and drop it into the Break/Tracepoints pane (see *Using the Break/Tracepoints pane* on page 5-59 for details).
- Set a breakpoint on a memory location in the Memory pane. The type of breakpoint offered depends on the type of memory at the chosen location, for example RAM, ROM, or Flash. See *Operating on memory contents* on page 7-17 for more details.
- Set a hardware breakpoint at an address of a symbol using the **Set Break At...** option on the Stack pane context menu, and assign a qualifier to the breakpoint. This displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).

However, this is not recommended if execution runs past the end of a function return call because as soon as you exit the function the stack value is no longer meaningful.

Use the Watch pane to track a specific symbol continuously because this recomputes the stack location dynamically and so tracks each invocation of the function.

You can use local variables within the conditional part of a breakpoint because the stack value is computed correctly each time the breakpoint condition is evaluated.

See *Operating on stack contents* on page 7-25 for more details on using the Stack pane.

- Right-click on a value shown in the Watch pane, or on a variable shown in the Call Stack pane, then select **Break at...** from the context menu. This displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).

See *Managing watches* on page 7-34 for more details on using the Watch pane.

See *Using the call stack* on page 7-28 for more details on using the Call Stack pane.

- You can use the Data Navigator pane to set a conditional breakpoint on a chosen function, defined as a location in the image. See *Using the Data Navigator pane* on page 9-5 for more details.

5.8 Using the Set Address/Data Breakpoint dialog box

The Set Address/Data Breakpoint dialog box provides comprehensive facilities to enable you to specify new breakpoints of any type in full (see *Breakpoint types* on page 5-2). You can also use it to edit existing breakpoints.

This section describes how to use the Set Address/Data Breakpoint dialog box, and includes:

- *Displaying the Set Address/Data Breakpoint dialog box*
- *Editing an existing breakpoint* on page 5-40
- *Set Address/Data Breakpoint dialog box interface components* on page 5-40
- *Setting software breakpoints* on page 5-43
- *Setting hardware breakpoints* on page 5-43
- *HW Support settings for target hardware* on page 5-45
- *Using ranges and masks with hardware breakpoints* on page 5-46
- *Specifying Qualifiers* on page 5-48
- *Specifying Actions* on page 5-49
- *Managing the qualifier and action lists* on page 5-51.

5.8.1 Displaying the Set Address/Data Breakpoint dialog box

To display the Set Address/Data Breakpoint dialog box, shown in Figure 5-13 on page 5-40, use one of the following methods:

- Select **Debug** → **Breakpoints** → **Set/Edit Breakpoint...** from the Code window main menu.
- Using the Break/Tracepoints pane, either:
 - select **Set/Edit Breakpoint...** from the **Pane** menu
 - right-click on the pane background, and select **Set/Edit Breakpoint...** from the context menu.
- Right-click on the margin, an instruction, or line of source code in the File Editor pane, and select **Set Break...** from the context menu.
- Right-click on a memory location in the memory pane, and select **Set Break At...** from the context menu.
- Right-click on a value shown in the Watch pane, or on a variable shown in the Call Stack pane, then select **Break at...** from the context menu.

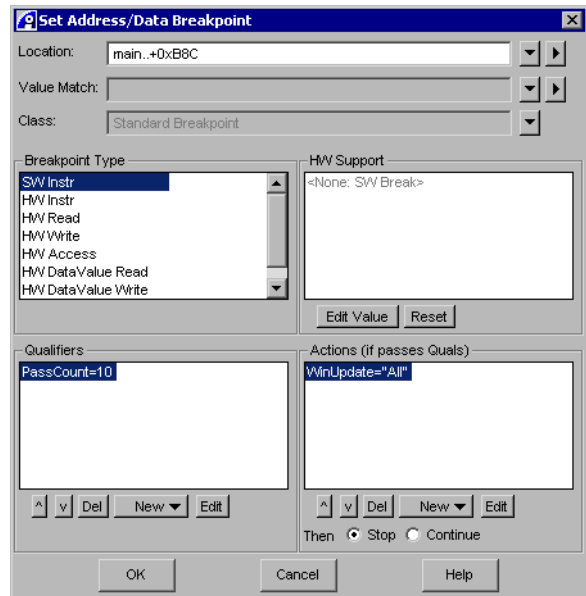


Figure 5-13 Set Address/Data Breakpoint dialog box

5.8.2 Editing an existing breakpoint

If you want to edit an existing breakpoint, right-click on the breakpoint entry in the Break/Tracepoints pane, and select **Edit Break/Tracepoint...** from the context menu. The Set Address/Data Breakpoint dialog box is displayed, and is populated with the current breakpoint attributes.

This is useful, for example, if you have previously set a default breakpoint, and you want to assign a condition qualifier or action to that breakpoint.

5.8.3 Set Address/Data Breakpoint dialog box interface components

The main interface components of the Set Address/Data Breakpoint dialog box (see Figure 5-13) are:

- Location** Specifies the memory location where the new breakpoint is set. This can be:
- a specific line number in the source code, with or without a module name prefix (see *Qualifying breakpoint line number references with module names* on page 5-6)
 - a specific address or address range (see *Specifying address ranges* on page 5-7)

- a function entry (see *Specifying the entry point to a function* on page 5-8).

You can enter the location directly in the field, or click the drop-down arrow to the right of this field to choose a location from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions. The options shown here depend on your debug target and connection.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see *Using ranges and masks with hardware breakpoints* on page 5-46 for details).

This field is enabled if you select a suitable Breakpoint Type and your current target supports the chosen type.

Value Match

Enter the data value that triggers the breakpoint. Alternatively, click the drop-down arrow to the right of this field to choose from a list browser (see *List browser dialog boxes* on page 9-2), select from your personal Favorites List (see *Favorites categories used by RealView Debugger features* on page 9-32), or select from a list of previously-used expressions.

If you use this with data breakpoints, this compares the data value that is read or written.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see *Using ranges and masks with hardware breakpoints* on page 5-46 for details).

This field is enabled if you select a suitable Breakpoint Type and your current target supports the chosen type.

Class

A read-only field to show the type of breakpoint you set.

By default, RealView Debugger sets a Standard Breakpoint but the contents of this field change depending on your debugging environment and target configuration. For example, the Class field might show Thread Breakpoint.

Breakpoint Type

Enables you to select the type of breakpoint to set. On first opening the dialog box, the list shows the breakpoint types that are supported by your debug target.

See Chapter 15 *RTOS Support* for more details on breakpoint classes, and how to work with different types of breakpoint.

If you select a Breakpoint Type, the contents of other groups in this dialog box might change. For example, if your current target supports hardware breakpoints, and you select a hardware breakpoint type, the HW Support group is populated with one or more options.

For details, see:

- *Setting software breakpoints* on page 5-43
- *Setting hardware breakpoints* on page 5-43.

HW Support

If your current target supports breakpoint tests in hardware, and you select a hardware Breakpoint Type, this group is populated with a list of currently available tests.

For details on using these controls, see:

- *HW Support settings for target hardware* on page 5-45.

Qualifiers

When setting a conditional breakpoint, you specify the condition that must be satisfied to trigger the breakpoint. Qualifiers are the tests that RealView Debugger carries out to trigger the breakpoint. If you specify more than one qualifier, then all those specified must be met before the breakpoint triggers. Click **New** to display the **New Qualifiers** menu, where you can define the test criteria. This menu contains the following options:

- **SW Pass Count...**
- **When Expression True...**
- **When Expression False...**
- **User Macro...**
- **C++ Object...**
- **Favorites...**

Any qualifiers you have set previously are listed below the **Favorites...** option.

See *Specifying Qualifiers* on page 5-48 for details on using these controls.

Actions

When a breakpoint triggers the usual action is to stop execution but you can specify one or more debugger actions that must be performed when execution stops. In addition, RealView Debugger can carry out the specified action and then execution can continue. This is useful when debugging complex applications without direct user intervention.

Click **New** to display the **New Actions** menu. This menu contains the following options:

- **Update Window...**
- **Update All Windows**
- **Update Sample...**
- **Breakpoint Timer**
- **Message Output...**
- **Favorites...**

Any actions you have set previously are listed below the **Favorites...** option.

See *Specifying Actions* on page 5-49 for details on using these controls.

- OK** Click **OK** to confirm the new breakpoint properties and close the dialog box.
- Cancel** Click **Cancel** to close the dialog box and abandon the breakpoint setting.
- Help** Click **Help** to get online help on the controls in this dialog box.

———— **Note** —————

Depending on the Breakpoint Type you select, the Location or Value Match fields might be unavailable. In this case, the field is grayed out. Also, in this example, the Class field is read-only and the drop-down arrow is unavailable.

5.8.4 Setting software breakpoints

To set a software breakpoint, select **SW Instr** as the Breakpoint Type. You can add qualifiers and actions as required. For more details, see:

- *Specifying Qualifiers* on page 5-48
- *Specifying Actions* on page 5-49.

5.8.5 Setting hardware breakpoints

Select a hardware Breakpoint Type to display entries specific to the hardware support for breakpoints available on the current target:

- HW Instr** Sets or modifies an instruction address breakpoint. This type of breakpoint enables you to perform hardware tests or to compare data values.

HW Read Sets or modifies a read breakpoint at the specified memory location or address range (see *Specifying address ranges* on page 5-7). The breakpoint is triggered if the application reads from any part of the specified memory range. Where supported by your debug target, you can also add data tests.

HW Write Sets or modifies a write breakpoint at the specified memory location or address range (see *Specifying address ranges* on page 5-7). The breakpoint is triggered if the application writes to any part of the specified memory range.

HW Access Sets or modifies an access breakpoint at the specified memory location or address range (see *Specifying address ranges* on page 5-7). The breakpoint is triggered when a memory address is accessed. This type of breakpoint enables you to perform hardware tests or to compare data values.

HW Data Value Read

Sets or modifies a breakpoint that is triggered if a specified data value is read from any address, and then detected by the debug hardware on the target processor.

HW Data Value Write

Sets or modifies a breakpoint that is triggered if a specified data value is written to any address, and then detected by the debug hardware on the target processor.

HW Data Value Access

Sets or modifies a breakpoint that is triggered if a specified data value is accessed at any address, and then detected by the debug hardware on the target processor.

For each of these types, use the HW Support group to specify how the match is configured, see:

- *HW Support settings for target hardware* on page 5-45

To set up the hardware breakpoint you must enter:

Location: Specifies the memory location where the new breakpoint is set. Where supported by your target hardware, use the options from the right-arrow menu to qualify the location (see *Using ranges and masks with hardware breakpoints* on page 5-46 for details).

Value Match:

Enter the data value that triggers the breakpoint.

If you use this with data breakpoints, this compares the data value that is read or written. When used with an instruction hardware breakpoint, this can test the value of an instruction. That is, it can be used to find an instruction in a given range.

Where supported by your target hardware, use the options from the right-arrow menu to qualify the value match (see *Using ranges and masks with hardware breakpoints* on page 5-46 for details).

Note

Depending on the Breakpoint Type you select, the Location or Value Match fields might be unavailable. In this case, the field is grayed out. Also, in this example, the Class field is read-only and the drop-down arrow is unavailable.

You can add qualifiers and actions as required. For more details, see:

- *Specifying Qualifiers* on page 5-48
- *Specifying Actions* on page 5-49.

5.8.6 HW Support settings for target hardware

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:

DataSize	Supports testing MAS signals from the core. This enables you to test the size of <i>data bus</i> activity.
Mode	Supports testing nTrans signals from the core. This enables you to test the <i>data not translate</i> signal to differentiate access between a User mode and a privileged mode.
Extern	Supports hardware breakpoints that are dependent on some external condition.

The current test is shown in round brackets, for example Ignore or Any.

To change a selected test, highlight the test, for example Match=Mode: (Ignore), and then click **Edit Value** to change how the test is defined, shown in Figure 5-14 on page 5-46.

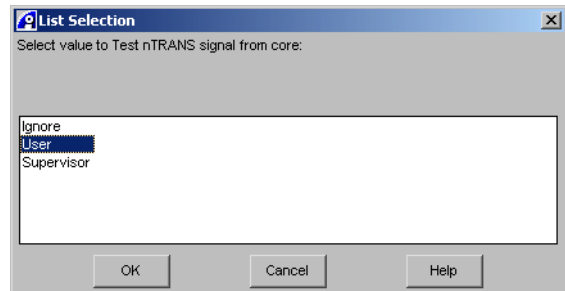


Figure 5-14 Configuring hardware tests

Click **OK** to confirm the new test and close the list selection box. The HW Support display list shows the new test.

Highlight a test and click **Reset** to restore the default settings.

5.8.7 Using ranges and masks with hardware breakpoints

Where supported by your target hardware, you can qualify the location and value match entries using options available from the right-arrow 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:

- **Address Range (HW Breaks only)**
- **Address Range by Length (HW Breaks only)**
- **Address Mask (HW Breaks only)**
- **NOT Address Compare (HW Breaks only)**
- **Autocomplete Range.**

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:

- **Value Range (HW Breaks only)**
- **Value Range by Length (HW Breaks only)**
- **Value Mask (HW Breaks only)**
- **NOT Value Compare (HW Breaks only)**
- **Autocomplete Range.**

Note

These menu options are only available where supported by your debug target and if you have specified a hardware Breakpoint Type.

Choose from the available options to set up your hardware breakpoint:

Address/Value Range

Enter the start address, or data value, for the breakpoint then click this option to specify a range, for example the address range 0x800FF..0x80A00 (see *Specifying address ranges* on page 5-7). The separators .. are automatically inserted for you.

Address/Value Range by Length

Enter the start address, or data value, for the breakpoint then click this option to specify a range by length, for example the address range 0x800FF..+0x1111 (see *Specifying address ranges* on page 5-7). The separators ..+ are automatically inserted for you.

Address/Value Mask

Enter the address, or data value, for the breakpoint then click 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 specified address, or data value, for example given the location 0x0111 and a mask 0x1001 the result is 0x0001.

The breakpoint triggers when the address, or data value, matches the given value after masking.

NOT Address/Value Compare

Enter the address, or data value, for the breakpoint then click 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

Enter a symbol and then click this option to compute the end-of-range address based on the symbol size. For example, if you enter a function then the autocompleted range is from the start to the end of the function. Similarly, enter a global variable to see the end-of-range address autocompleted as the variable storage address plus variable size.

Note

Many combinations of range, or mask, options are permitted. However, mixing range and mask generates a warning message to say that this is not permitted.

5.8.8 Specifying Qualifiers

To set a conditional breakpoint, you must assign a condition qualifier. To do this, click the **New** button in the Qualifiers group, shown in Figure 5-15, to display the **New Qualifiers** menu. You can use this to specify qualifiers when you first create a breakpoint, or to add qualifiers to edit an existing breakpoint.

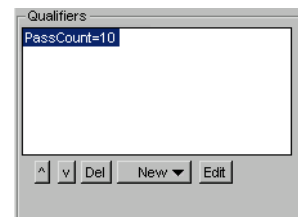


Figure 5-15 Assigning qualifiers to breakpoints

The order of the qualifiers, in the Qualifiers group display list, defines the order they are tested to trigger the breakpoint. If a condition is False, then subsequent conditions are not tested. See *Controlling the behavior of breakpoints* on page 5-55 for details on how the order of conditional qualifiers affects the behavior of a breakpoint.

The New Qualifiers menu

This menu enables you to select the qualifier that controls execution, that is to define the condition that must be satisfied to trigger the breakpoint:

SW Pass Count...

Use this to specify the number of times execution must arrive at the specified address before execution stops.

The equivalent CLI command qualifier is `passcount:count`.

When Expression True...

Enables you to specify an expression that must evaluate to True to stop execution.

The equivalent CLI command qualifier is `when:{condition}`.

When Expression False...

Use this to specify an expression that must evaluate to False to stop execution.

The equivalent CLI command qualifier is `when_not:{condition}`.

User Macro...

Enables you to specify a macro that runs when execution stops. This brings up a dialog box where you supply the macro name and any arguments required to run it. See *Attaching macros to breakpoints* on page 5-52 for an example. Also, see *Controlling the behavior of breakpoints* on page 5-55 for details on how the macro return value affects the behavior of a breakpoint.

The equivalent CLI command qualifier is `macro:{MacroCall(arg1,arg2)}`.

C++ Object...

Use this to specify a C++ this object to test. The Call Stack pane contains a **This** tab where you can view such objects.

The equivalent CLI command qualifier is `obj:(n)`.

Favorites... Select this option to display your personal Favorites List of breakpoint qualifiers (see *Favorites categories used by RealView Debugger features* on page 9-32). From here you can specify the required qualifier.

See *Setting breakpoints from your breakpoint Favorites List* on page 5-71 for details of creating breakpoint favorites and adding qualifiers to the list.

5.8.9 Specifying Actions

You can assign actions to your breakpoint. To do this, click the **New** button in the Actions group, shown in Figure 5-16, to display the **New Actions** menu. You can use this to specify breakpoint actions when you first create a breakpoint, or to add actions to edit an existing breakpoint.

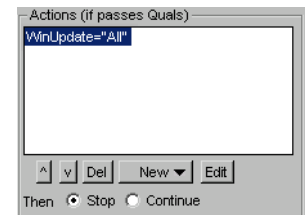


Figure 5-16 Assigning actions to breakpoints

The order of the actions, in the Actions group display list, defines the order they are carried out when the breakpoint triggers.

Actions do not execute until all condition qualifiers, if any, complete successfully. If no condition qualifiers are specified, the actions are always performed. See *Controlling the behavior of breakpoints* on page 5-55 for details on how actions are performed for a breakpoint when used with condition qualifiers.

The New Actions menu

The **New Actions** menu enables you to specify one or more actions to be performed when a breakpoint triggers, and any condition qualifiers, if any, complete successfully:

Update Window...

Displays a list selection box where you can choose which debugger windows are updated when execution stops. The list includes all windows and panes available from the default desktop. You can also redirect debugger output to a user window and this can be updated when execution stops.

You can only use this selection box to specify one window at a time. If you want to update several windows, repeat the operation to set up a windows list. The list order specifies the update order.

The equivalent CLI command qualifier is `update:{"name"}`.

Update All Windows

Updates all desktop windows when the breakpoint triggers.

The equivalent CLI command qualifier is `update:{"all"}`.

Update Sample...

Updates registered samples when execution stops. This is used as part of the graphing and visualization functions in RealView Debugger.

————— **Note** —————

This menu option is not available for visualization functions in this release. This option is available when a sampling variant is available, for example logging from breaks.

Breakpoint Timer

This option is enabled if your debug target supports cycle timing in hardware. The timer measures execution time from the point where the breakpoint triggers.

The equivalent CLI command qualifier is `timed`.

Message Output...

Displays a dialog box where you can enter a short text string for display when the breakpoint triggers.

By default this message appears in the Output pane but you can specify a the ID of a user defined window or an open file. For example, if you have a user defined window with an ID of 250, then `250Stop at convert proc` sends the message `Stop at convert proc` to that window (see *Attaching macros to breakpoints* on page 5-52 for an example).

The equivalent CLI command qualifier is `message:{"$windowid | fileid$message"}`.

Favorites... Select this option to display your personal Favorites List of breakpoint actions (see *Favorites categories used by RealView Debugger features* on page 9-32). From here you can specify the required action.

If you have used actions previously, these are displayed at the bottom of the **New Actions** menu.

Setting the continuation state

Use the **Stop** and **Continue** options to specify whether the program is to continue or stop executing when the breakpoint is triggered. The default state is to stop execution. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.

The **Continue** option is equivalent to the `continue` command qualifier of the breakpoint CLI commands. The **Stop** option does not have an equivalent command qualifier.

See *Controlling the behavior of breakpoints* on page 5-55 for details on how the continuation state affects the behavior of a breakpoint.

5.8.10 Managing the qualifier and action lists

To manage the Qualifiers and Actions display lists:

- re-order the list by highlighting a qualifier or action, and use the up or down button to reposition the specified entry in the list
- highlight a qualifier or action, and click **Del** to delete the specified entry
- highlight a qualifier or action, and click **Edit** to update it so changing the behavior.

Qualifiers, actions, and the continuation state, are set up using the Set Address/Data Breakpoint dialog box, shown in Figure 5-13 on page 5-40.

5.9 Attaching macros to breakpoints

RealView Debugger includes a macro facility that enables you to create macros containing complex procedures. You can execute these macros directly through the RealView Debugger CLI, or you can attach a macro to a breakpoint so that it is executed when the breakpoint triggers. When you attach a macro to a breakpoint, the macro can return an integer value that determines whether program execution continues or stops. See *Controlling the behavior of breakpoints* on page 5-55 for details on how the macro return value affects the behavior of a breakpoint.

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, for example the `tutorial()` macro created in *Using macros* on page 10-8, then this must be defined as a symbol in the debugger.

Macro syntax is defined in the *RealView Developer Kit v2.2 Command Line Reference*.

———— Note ————

Execution-type commands are not valid within a breakpoint macro. See *Using macros with breakpoints* on page 10-6 for full details.

This section includes:

- *Defining a user-created macro as a symbol*
- *Attaching a macro to a breakpoint* on page 5-53.

5.9.1 Defining a user-created macro as a symbol

To define a user-created macro as a symbol, so that you can attach it to a breakpoint:

1. Make sure the macro is defined in an include file, for example `tutorial.inc`.
2. Select **Tools** → **Include Commands from File...** to display the Select File to Include Commands from dialog box.
3. Locate and highlight the macro file. That is, `tutorial.inc` in this example.
4. Click **Open** to load the macro file into the debugger. The debugger defines the macro as a symbol with the same name as the macro, for example `tutorial`.

5.9.2 Attaching a macro to a breakpoint

You can attach a predefined macro, or a macro you have created yourself, to a breakpoint. If you want to attach a macro you have created yourself, you must first load the macro definition into the debugger as described in *Defining a user-created macro as a symbol* on page 5-52.

To attach a macro to a breakpoint:

1. Display the Set Address/Data Breakpoint dialog box using one of the methods described in *Displaying the Set Address/Data Breakpoint dialog box* on page 5-39.
2. Enter the location of the breakpoint, for example 0x8704.
3. Click the **New** button in the Qualifiers group to display the **New Qualifiers** menu.
4. Select the option **User Macro...** to display the data entry prompt, shown in Figure 5-17.

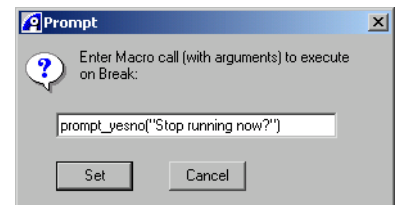


Figure 5-17 Breakpoint macro entry prompt

5. Enter the macro name and arguments, if any.
The predefined macro shown in Figure 5-17 displays a message box to halt execution if the **Yes** key is pressed.
6. Click **Set** to confirm your entry.
7. The Set Address/Data Breakpoint dialog box now contains the macro in the Qualifiers group for the conditional breakpoint, shown in Figure 5-18.

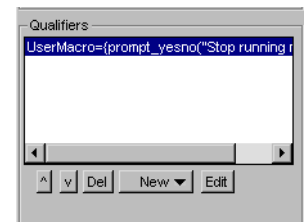


Figure 5-18 Set breakpoint with macro attached

8. Click **OK** to confirm the breakpoint settings and close the dialog box.
9. Click **Run** to execute the program and trigger the breakpoint.

5.10 Controlling the behavior of breakpoints

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.

This section includes:

- *Demonstrating breakpoint behavior with the Dhrystone example project*
- *Using a macro as an argument to a break command on page 5-57.*

5.10.1 Demonstrating breakpoint behavior with the Dhrystone example project

To demonstrate the behavior of a breakpoint using the Dhrystone example project:

1. Load the `dhrystone.prj` project file, and rebuild the image if necessary.
2. Connect to a debug target.
3. Load the `dhystone.axf` image.
4. Set a software breakpoint in `dhry_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 (see the chapter that describes the CLI commands in the *RealView Developer Kit v2.2 Command Line Reference*).

Use the following command:

```
breakinstruction,qualifiers (I A)\DHR_1\#153:2
```

For *qualifiers* use the following (see the command qualifiers in Table 5-3 on page 5-56, for each command qualifier variation):

- A macro to view a symbol, for example:

```
define /R int viewSymbol()
{
    $printsymbols DHR_1\clock_t$;
    return (0); // 0 - stop execution at the breakpoint
              // >=1 - continue execution
}
```

See Chapter 10 *Working with Macros* for more details on defining and using macros. Also, see *Using a macro as an argument to a break command on page 5-57.*

- A pass count specifying five passes before the breakpoint is to be triggered.

- An action qualifier that prints the message `Actions performed`.
5. Run the image, enter 10 when prompted for the number of runs.
 6. If you are using the RealView Debugger GUI, click the **Cmd** tab in the Output pane to view the results.
 7. Repeat steps 3 to 6 for each command qualifier variation, and change the macro return value as indicated.

Table 5-3 summarizes the breakpoint behavior for various combinations of the command qualifiers.

Table 5-3 Breakpoint behavior with multiple condition qualifiers and actions

Command qualifiers	Macro return value	Execute macro	Print message	Record the breakpoint details
,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

Table 5-3 Breakpoint behavior with multiple condition qualifiers and actions

Command qualifiers	Macro return value	Execute macro	Print message	Record the breakpoint details
,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 0x000084C4 due to SW Instruction Breakpoint
Stopped at 0x000084C4: DHRV_1\main Line 153
```

From Table 5-3 on page 5-56 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.

5.10.2 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"} (I A)\DHR_1\#153:2
;viewSymbol()
```

This command gives the same behavior where the command qualifiers are in the following order (see Table 5-3 on page 5-56):

```
,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)\DHR_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.

5.11 Using the Break/Tracepoints pane

The Break/Tracepoints pane provides a central point to manage all breakpoint operations. It enables you to:

- view all breakpoints currently set and their status (enabled or disabled)
- specify the command used to create a breakpoint
- edit an existing breakpoint specification, for example you might want to change the location, update the qualifiers, or change debugging actions when the breakpoint triggers
- copy the attributes of an existing breakpoint to create a new breakpoint at another location
- create new breakpoints as an alternative to using the **Debug** menu
- access the **Break/Tracepoints** and **Pane** menus to manage your breakpoint operations.

Note

The Break/Tracepoints pane also enables you to manage tracepoints during Trace and Analysis operations, and to work with thread breakpoints when running in RSD mode.

This section describes:

- *Displaying the Break/Tracepoints pane* on page 5-60
- *Viewing breakpoints in the Break/Tracepoints pane* on page 5-61
- *Using the Break/Tracepoints Entry context menu* on page 5-62
- *Using the Break/Tracepoints Entry context menu* on page 5-62
- *Working in the Break/Tracepoints pane* on page 5-64
- *Using the Pane menu* on page 5-65.

5.11.1 Displaying the Break/Tracepoints pane

Select **View** → **Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 5-19.

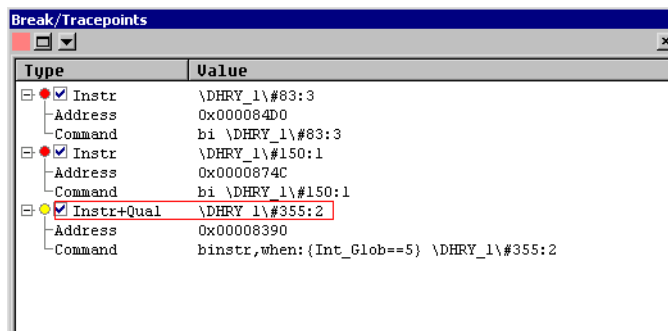


Figure 5-19 Break/Tracepoints pane

If any breakpoints are set already, these are displayed in the new pane. To expand the tree and see the breakpoint details either:

- Click on the plus sign associated with a particular breakpoint entry.
- Double-click anywhere along the breakpoint entry (not on the check box).

The Break/Tracepoints pane also displays any other breakpoints you set, for example thread breakpoints.

If no breakpoints are set then the Break/Tracepoints pane is empty. As you create and set new breakpoints, the pane is automatically updated to display each new breakpoint.

5.11.2 Viewing breakpoints in the Break/Tracepoints pane

The Break/Tracepoints pane shows entries in a tree view giving the Type and Value of each breakpoint you set, shown in the example in Figure 5-20.

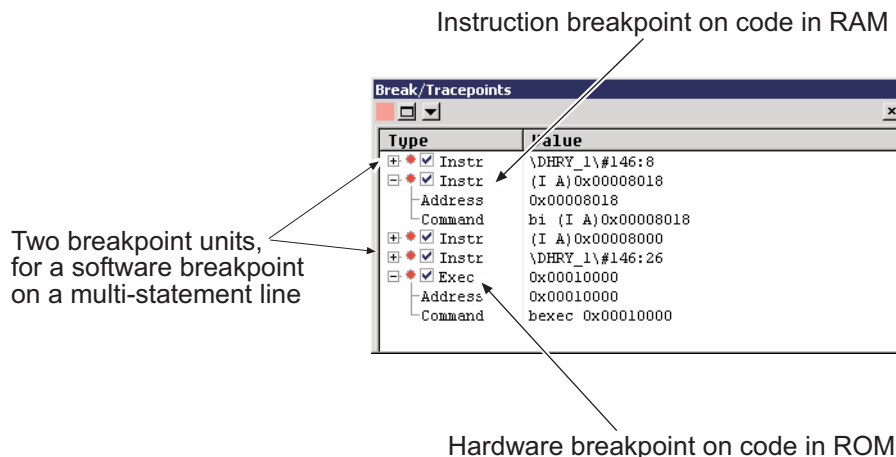


Figure 5-20 Breakpoints in the Break/Tracepoints pane

Each breakpoint is identified by a:

- colored icon to show the breakpoint type (see *Viewing breakpoints in your code view* on page 5-10 for details)
- check box to show if the breakpoint is enabled or disabled. You can click on the check box to change the state of the chosen breakpoint or breakpoint unit.

In Figure 5-20, an area of memory on the debug target has been designated ROM and so a BREAKEXECUTION command is used to set a hardware breakpoint. When the hardware breakpoint limit is reached for this debug target no more breakpoints can be set on code in this area, and a warning message is displayed.

———— Note ————

For details on the BREAKINSTRUCTION and BREAKEXECUTION commands see RealView Developer Kit v2.2 Command Line Reference.

In the example in Figure 5-21 on page 5-62, different types of hardware breakpoints have been set, and disabled, during the session.

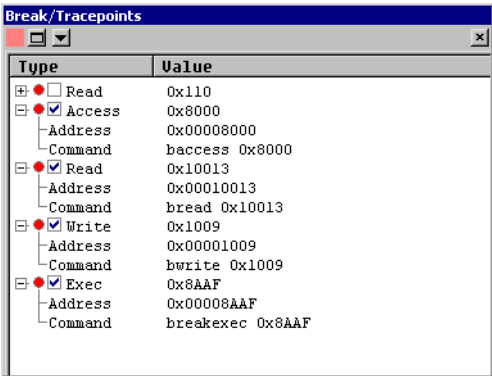


Figure 5-21 Hardware breakpoints in the Break/Tracepoints pane

5.11.3 Using the Break/Tracepoints Entry context menu

If you have a breakpoint set, right-click anywhere along the breakpoint entry to display the **Break/Tracepoints Entry** context menu.

The **Break/Tracepoints Entry** context menu includes the following options to edit and manage your breakpoints:

Edit Break/Tracepoint...

For breakpoints, displays the Set Address/Data Breakpoint dialog box, where you can edit the breakpoint details (see *Editing a breakpoint* on page 5-64).

Copy Break/Tracepoint...

For breakpoints, displays the Set Address/Data Breakpoint dialog box with details of the original breakpoint (see *Copying an existing breakpoint* on page 5-65).

Clear Removes the breakpoint selected in the Break/Tracepoints pane.

Disable This option changes the status of a selected breakpoint.

Select **Disable** to disable a breakpoint you have set. This changes to **Enable** so that you can enable a breakpoint that has been disabled.

Select, or unselect, the check box in the Break/Tracepoints pane to change the status in the same way.

Reset PassCounters/Then-Enables

Resets the counters that are used to register the number of times that the program passes a breakpoint before it is triggered. If the breakpoint is enabled, then it stops being triggered until the count value is reached again.

Details... Displays an information box giving details of the selected breakpoint.

Break/Tracepoint Favorites...

Displays the Favorites Chooser/Editor where you can edit or delete entries, or select from your favorites to set a breakpoint (see *Favorites categories used by RealView Debugger features* on page 9-32).

Show Code The cursor moves to the location of the breakpoint in the appropriate code view:

- an open blue pointer marks the location in source-level view
- a filled blue pointer marks the location in disassembly-level view if there is no source.

5.11.4 Using the Break/Tracepoints context menu

Right-click anywhere on the background of the Break/Tracepoints pane to display the **Break/Tracepoints** context menu. This context menu includes the following options to edit and manage your breakpoints:

Set Break from Function/Label list

Displays the Function Breakpoint/Profile Selector dialog box (see *Setting breakpoints from saved lists* on page 5-69).

Set/Edit Breakpoint...

Displays the Set Address/Data Breakpoint dialog box (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).

Set BreakIf...

Displays a List Selection dialog box containing a list of the breakpoints you can set. Each option displays a dialog box to set the breakpoint, as described in:

- *Setting unconditional breakpoints explicitly* on page 5-20
- *Setting hardware breakpoints explicitly* on page 5-22
- *Setting conditional breakpoints* on page 5-32.

Set Tracepoint...

Displays the Set/Edit Tracepoint dialog box.

Processor Exceptions...

Displays a List Selection dialog box containing a list of processor exceptions that you can enable or disable.

Break/Tracepoint History...

Displays a List Select dialog containing a list of breakpoints that you have previously set (see *Setting breakpoints from saved lists* on page 5-69).

Break/Tracepoint Favorites...

Displays a List Select dialog containing a list of breakpoint favorites that you have previously defined (see *Favorites categories used by RealView Debugger features* on page 9-32 and *Setting breakpoints from saved lists* on page 5-69).

Show Break Capabilities of HW...

Displays information about the hardware breakpoint support on the current debug target.

Clear All Break/Tracepoints

Clears all breakpoints defined for the current debug target.

5.11.5 Working in the Break/Tracepoints pane

If you have a breakpoint set, you can use the Break/Tracepoints pane to edit or copy a breakpoint, as described in:

- *Editing a breakpoint*
- *Copying an existing breakpoint* on page 5-65.

You can also use the Break/Tracepoints pane to edit or copy other breakpoints you set, for example thread breakpoints.

Editing a breakpoint

To edit a breakpoint:

1. Right-click on a breakpoint in the list to display the **Break/Tracepoints Entry** menu (see *Using the Break/Tracepoints Entry context menu* on page 5-62).
2. Select **Edit Break/Tracepoint...** from the context menu to display the Set Address/Data Breakpoint dialog box where you can edit the breakpoint or breakpoint unit (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).

When you edit a breakpoint, the breakpoint command includes the `modify` command qualifier (see the *RealView Developer Kit v2.2 Command Line Reference* for more details).

Copying an existing breakpoint

To copy an existing breakpoint and create a new breakpoint:

1. Right-click on a breakpoint in the list to display the **Break/Tracepoints Entry** menu (see *Using the Break/Tracepoints Entry context menu* on page 5-62).
2. Select **Copy Break/Tracepoint...** from the context menu to display the Set Address/Data Breakpoint dialog box, populated with the relevant information about the chosen breakpoint (see *Using the Set Address/Data Breakpoint dialog box* on page 5-39).
3. Change the breakpoint location, to create the new breakpoint.

5.11.6 Using the Pane menu



Click on the drop-down arrow on the Break/Tracepoints pane toolbar to display the **Pane** menu. This menu includes options that are also available from the **Debug** menu. The relationship between the two menus is summarized in Table 5-4.

Table 5-4 Breakpoint menu mappings

Break/Tracepoints pane, Pane menu	Code window, Debug menu
Set Break from Function/Label list	Breakpoints → Set Break/Tracepoint from List → Set from Function/Label list...
Set/Edit Breakpoint...	Breakpoints → Set/Edit Breakpoint...
Set BreakIf...	Breakpoints → Conditional and Breakpoints → Hardware
Set Tracepoint...	Tracepoints → Set Tracepoint...
Processor Exceptions...	Processor Exceptions...
Break/Tracepoint History...	Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint History...

Table 5-4 Breakpoint menu mappings

Break/Tracepoints pane, Pane menu	Code window, Debug menu
Break/Tracepoint Favorites...	Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...
Show Break Capabilities of HW...	Breakpoints → Hardware → Show Break Capabilities of HW...
Clear All Break/Tracepoints	Breakpoints → Clear All Break/Tracepoints

If you select the option **Set BreakIf...** from the **Pane** menu, a breakpoint type selection box is displayed containing all the breakpoints supported by your debug target. Each entry displays a dialog box to set the breakpoint, as described in:

- *Setting unconditional breakpoints explicitly* on page 5-20
- *Setting hardware breakpoints explicitly* on page 5-22
- *Setting conditional breakpoints* on page 5-32.

You can also access some of the options from the **Pane** menu by right-clicking on a blank area inside the Break/Tracepoints pane. This enables you to create new breakpoints, select from your personal favorites or history list, or to see your hardware support.

5.12 Disabling and clearing breakpoints

You can temporarily disable breakpoints. This does not delete the breakpoint but means that you can enable it quickly for re-use in your current debugging session.

If you disable a breakpoint, you can still view it in the **Src** or **Dsm** tab where it is shown as a white disc. Any accompanying downward pointing arrows are colored light gray.

When you clear a breakpoint it is removed from the breakpoint list. To remove breakpoints from your Favorites List, use the Favorites Chooser/Editor dialog box, see *Setting breakpoints from your breakpoint Favorites List* on page 5-71 for details of how to do this.

This section describes:

- *Disabling breakpoints*
- *Clearing breakpoints* on page 5-68
- *Clearing all breakpoints* on page 5-68.

5.12.1 Disabling breakpoints

You can enable and disable breakpoints from the Code window:

- In the **Src** tab:
 1. Right-click in the left margin, or on the line number, of a line marked with a red breakpoint icon, or on the icon itself.
 2. Select **Disable Break** from the context menu.
- In the Break/Tracepoints pane, select the check box to unselect it so disabling the breakpoint.
- Right-click on the breakpoint in the Break/Tracepoints pane to display the **Break/Tracepoints** menu. Select **Disable** from the menu.
- Position the cursor at the breakpoint that you want to disable. Select **Debug** → **Breakpoints** → **Enable/Disable Break at Cursor** from the main menu.

You can use these methods to:

- enable a disabled breakpoint, marked by a white icon
- enable, or disable, a conditional breakpoint, marked by a yellow icon
- enable, or disable, thread breakpoints.

5.12.2 Clearing breakpoints

You can clear breakpoints from the Code window in different ways:

- In the **Src** tab, double-click on the line number of a line marked with a red breakpoint icon or on the icon itself.
- Right-click on the breakpoint in the Break/Tracepoints pane to display the **Break/Tracepoints** menu. Select **Clear** from the menu.
- Position the cursor at the breakpoint that you want to disable. Select **Debug → Breakpoints → Toggle Break at Cursor** from the main menu.

You can use these methods to clear any type of breakpoint, for example a conditional breakpoint (marked by a yellow icon), or a thread breakpoint.

5.12.3 Clearing all breakpoints

You can clear all the breakpoints set on a selected debug target in one operation from the Code window, either:

- Select **Debug → Breakpoints → Clear All Break/Tracepoints** from the main menu.
- Click on the **Pane** menu in the Break/Tracepoints pane and select **Clear All Break/Tracepoints**.

5.13 Setting breakpoints from saved lists

This section describes how to set breakpoints from various lists held by RealView Debugger. If you use specific breakpoint definitions on a regular basis, then you can also define your own breakpoint lists. This section includes:

- *Setting breakpoints from the Function/Label list*
- *Setting breakpoints from the breakpoint history lists* on page 5-70
- *Setting breakpoints from your breakpoint Favorites List* on page 5-71.

5.13.1 Setting breakpoints from the Function/Label list

You can set a breakpoint on any number of the function names and labels in your image. The list of function names and labels in your image is available on the Function Breakpoint/Profile Selector dialog box. To display this dialog box, use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Set Break/Tracepoint from List → Set from Function/Label list...
- Select the following option from the **Pane** menu of the Break/Tracepoints pane:
Set Break from Function/Label list...
- Right-click on the background of the Break/Tracepoints pane, and select the following option from the context menu:
Set Break from Function/Label list

Setting a breakpoint

To use the Function Breakpoint/Profile Selector dialog box to set a breakpoint on a chosen function:

1. Click on the check box for those functions where you want to set a breakpoint.
2. Click **Set** to set a breakpoint on the chosen functions.

Because the browser is used only to make a selection, there are no controls for debugging operations.

The Function Breakpoint/Profile Selector dialog box does not provide a record of breakpoints already set, that is, when you next open this dialog box existing breakpoints are not checked.

5.13.2 Setting breakpoints from the breakpoint history lists

Your personal history file, `exhist.sav`, is saved in your RealView Debugger home directory and is updated when you close down at the end of your session. It contains a snapshot of the current breakpoints across all your debug targets. The items in this list include breakpoints you have set during the current debugging session, and breakpoints you have set from previous debugging sessions.

To see the current breakpoint history, use one of the following methods:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Set Break/Tracepoint from List... → Break/Tracepoint History...
- Select the following option from the **Pane** menu of the Break/Tracepoints pane:
Break/Tracepoint History...
- Right-click on the background of the Break/Tracepoints pane, and select the following option from the context menu:
Break/Tracepoint History...

Breakpoints are only added to the history list if they are set using breakpoint dialog boxes, for example the Set Address/Data Breakpoint or the Simple Break if X dialog box. If you set a breakpoint in another way, for example using a CLI command, this is not added to the list. To force this type of breakpoint to be added to the history list:

1. Set the required breakpoint using the appropriate CLI command, for example:
`bi \DHRV_1\#153:1`
2. Select **View → Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane, shown in Figure 5-19 on page 5-60.
3. Right-click anywhere along the required breakpoint entry (not on the check box) to display the **Break/Tracepoints Entry** context menu (see *Using the Break/Tracepoints Entry context menu* on page 5-62).
4. Select **Edit Break/Tracepoint...** to display the Set Address/Data Breakpoint dialog box.
5. Click **OK** to close the dialog box without changing the breakpoint details.

See *Setting breakpoints from your breakpoint Favorites List* on page 5-71 for details of creating breakpoint favorites and adding existing breakpoints to your personal Favorites List.

5.13.3 Setting breakpoints from your breakpoint Favorites List

When you first start to use RealView Debugger on Windows, 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 to enable RealView Debugger favorites.

You create and edit breakpoint favorites using the Favorites Chooser/Editor dialog box. This section describes how to use this dialog box, and contains:

- *Displaying the Favorites Chooser/Editor dialog box*
- *Creating a new breakpoint favorite on page 5-72*
- *Saving existing breakpoints as favorites on page 5-72.*

Displaying the Favorites Chooser/Editor dialog box

Use one of the following methods to display the Favorites Chooser/Editor dialog box:

- Select the following option from the Code window main menu:
Debug → Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...
- Select the following option from the **Pane** menu of the Break/Tracepoints pane:
Break/Tracepoint Favorites...
- Right-click on the background of the Break/Tracepoints pane, and select the following option from the context menu:
Break/Tracepoint Favorites...

For full details on how to use the Favorites Chooser/Editor dialog box, see *Using the Favorites Chooser/Editor dialog box* on page 9-29.

Creating a new breakpoint favorite

To create a new breakpoint and add it to your Favorites List:

1. Display the Favorites Chooser/Editor dialog box as described in *Displaying the Favorites Chooser/Editor dialog box* on page 5-71.
2. Click **New** to display the New/Edit Favorite dialog box.
3. Enter the CLI command to create the breakpoint and, optionally, a short text description to identify the breakpoint for future use.
4. 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.
5. Use the available controls to update the new breakpoint favorite (see *Favorites Chooser/Editor dialog box interface components* on page 9-30).

Saving existing breakpoints as favorites

If you have already set some breakpoints, RealView Debugger lets you choose which to add to your Favorites List so that they are available to re-use in future debugging sessions or with other build target configurations of your application program.

To add existing breakpoints to your Favorites List:

1. Select **View** → **Break/Tracepoints** to display the Break/Tracepoints pane.
2. Highlight a breakpoint in the display list that you want to add to your Favorites List.
3. Right-click and select **Break/Tracepoint Favorites...** from the **Break/Tracepoints** menu. This displays the Favorites Chooser/Editor dialog box with the breakpoint command included in the **Add to List** field.
4. Click **Add to List** to add the specified breakpoint to your Favorites List. This displays the New/Edit Favorite dialog box.
The Expression field contains the breakpoint details. The Description field enables you to enter a short text description to help you to identify the breakpoint. This text is optional.
5. Click **OK** to confirm the breakpoint details and close the dialog box.

The Favorites Chooser/Editor dialog box is updated to show the new breakpoint in the display list. Because this breakpoint is already set, click **Close** to close the dialog box. If required, you can set another breakpoint from your Favorites List before closing the dialog box.

The edited Favorites List is saved to your `exphist.sav` file, in your home directory, when you close RealView Debugger.

Chapter 6

Memory Mapping

This chapter describes how to manage target memory in the RealView® Debugger Process Control pane. It contains the following sections:

- *About memory mapping* on page 6-2
- *Enabling and disabling memory mapping* on page 6-4
- *Setting up a memory map* on page 6-5
- *Viewing the memory map* on page 6-7
- *Editing map entries* on page 6-12
- *Setting top_of_memory and stack values* on page 6-13
- *Generating linker command files for non-ARM targets* on page 6-14.

6.1 About memory mapping

Memory mapping is disabled by default when you first connect to your debug target, that is all memory is treated as RAM. If you are working with a suitable target you can enable memory mapping and then configure the memory using the **Map** tab in the Process Control pane.

This section includes:

- *Uses for memory mapping*
- *Memory map considerations* on page 6-3.

6.1.1 Uses for memory mapping

Memory mapping might be useful depending on the debug target you are using and the applications you are developing:

- If you are working with a simulator, or development board, to develop your application program, mapping memory ensures that you are not trying to load your image into memory that does not exist on the real hardware.
- If you are working with different types of memory, memory mapping enables you to load an image and specify the exact location of different sections of memory.
- If your application contains pointers or stacks, or other uses outside declared areas, it might not work correctly on the final hardware. Mapping memory as Auto makes these errors visible.
- You can declare and program Flash memory with appropriate support files.
- Memory mapping can be used to generate linker information when developing a scatterloaded application. This is not supported by ARM® architecture-based targets.
- Memory mapping enables RealView Debugger to handle shared memory so that references are updated when modified by a different processor. This also prevents software breakpoints in shared code memory.
- If you are using a simulator that supports it, you can map memory to add wait states to obtain better cycle accuracy when profiling or measuring performance.
- Like register modeling, memory mapping provides a means for system developers to tell users about the memory configuration on boards, chips or in simulator models.

6.1.2 Memory map considerations

When working with memory mapping, you must be aware of the following:

- The `top_of_memory` value must be higher than the sum of the program base address and program size. If set incorrectly, your program might crash because of stack corruption or because your program overwrites its own code.
- There is no requirement that the address specified by `top_of_memory` is at the true top of memory. A C or assembler program can use memory at higher addresses.
- If you are working with a scatterloaded application, you must define the location of stack and heap in your code.
- When an image is loaded to a debug target, the memory map is checked 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.
- By default, RealView Debugger tries to set a software breakpoint. However, where enabled, the memory map determines the type of breakpoint that you can set. For example, only hardware breakpoints can be set in Flash and ROM.
- The memory map is used to define how memory contents are color coded when displayed in the Memory pane.

6.2 Enabling and disabling memory mapping

To enable memory mapping before you load an image:

1. Connect to a suitable target.
2. Select **View** → **Memory Map tab** to display the Process Control pane and bring the **Map** tab to the front. The pane is displayed as a floating pane.
Alternatively, if you have an instance of the Process Control pane that is docked, select the **Map** tab in that pane.
3. Right-click anywhere in the **Map** tab to display the **Map** context menu (see *Using the Map tab context menu* on page 6-10).
4. Click on the option **Toggle Memory Mapping** so that it is checked.
This enables memory mapping ready to load your image.
5. To disable memory mapping, right-click anywhere in the **Map** tab to display the **Map** context menu.
The context menu contains different options depending on the state of your debug target. For example, if you load an image the menu contains the option **Update Map based on Image**.
6. Click on the option **Toggle Memory Mapping** so that it is unchecked.
This disables memory mapping ready to load your image.
See *Using the Map tab context menu* on page 6-10 for details on the other options available from this menu.

Note

If you have assigned a board-chip definition file to a target connection, and that board-chip definition defines a memory map, then memory mapping is automatically enabled when you connect to that target.

6.3 Setting up a memory map

Mapping memory, before you load an image for debugging, enables you to have full access to all the memory on your debug target. You can do this:

- as part of your target configuration settings
- using an include file
- interactively using the **Map** tab
- by submitting the appropriate CLI commands.

In this example, you are going to set up memory manually for the current session. Target memory settings defined in this way are only temporary and are lost when you close down RealView Debugger.

With memory mapping enabled, set up your map in the Process Control pane:

1. Right-click on the Start entry in the **Map** tab to display the **Map** context menu (see *Using the Map tab context menu* on page 6-10).
2. Select **Add or Copy Map Entry...** to display the Add/Copy/Edit Memory Map dialog box, shown in Figure 6-1.

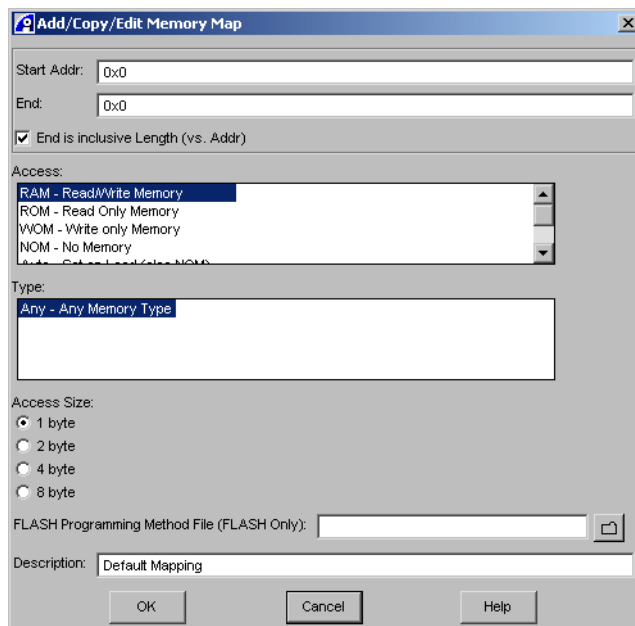


Figure 6-1 Add/Copy/Edit Memory Map dialog box

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 unselect the **End is inclusive Length (vs Addr.)** check box.

3. Edit the dialog box to change the default memory mapping as follows:
 - a. Enter 0x0 in the Start Addr field.
 - b. Enter 0x8000 in the End field to specify the block size.
4. Enter Area before image in the Description field to describe this block.
5. Click **OK** to confirm your changes and the Process Control **Map** tab is updated.
6. Set up the second block of memory using these settings:
 - a. Start address = 0x8000
 - b. Length = 0x8000
 - c. Description = Middle
7. Click **OK** to confirm your changes and the Process Control **Map** tab is updated.
8. Set up the third block of memory using these settings:
 - a. Start address = 0x10000
 - b. Length = 0xFFFF0000
 - c. Description = Area after image
9. By default, memory access is set to byte-size (8 bits) for ARM processors. Do not change this.
10. Click **OK** to confirm your changes and update the Process Control **Map** tab, shown in Figure 6-2.

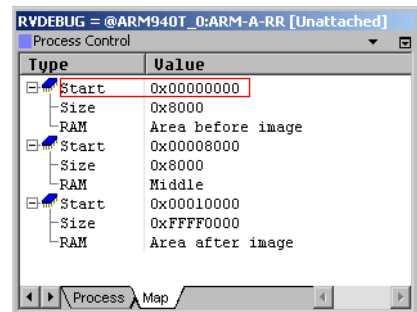


Figure 6-2 Memory mapped

6.4 Viewing the memory map

The Process Control pane provides a view of the memory mapping for the debug target that is running your application. This section describes how to use the map:

- *Working with the Map tab*
- *Memory map configuration* on page 6-8
- *Using the Map tab context menu* on page 6-10.

6.4.1 Working with the Map tab

To view the memory map:

1. Connect to a suitable debug target.
2. Enable memory mapping (see *Enabling and disabling memory mapping* on page 6-4).
3. Load an image, for example `dhrystone.axf`
4. Select **View** → **Memory Map tab** to display the Process Control pane and bring the **Map** tab to the front. The pane is displayed as a floating pane.

Alternatively, if you have an instance of the Process Control pane that is docked, select the **Map** tab in that pane.
5. Click on the plus signs to expand the entries, shown in Figure 6-3 on page 6-9.

The **Map** tab displays a tree-like structure for each component of the memory map showing the start address, size, and access rule. A one-line text description can also be included. The way that memory is shown depends on your debug target because RealView Debugger populates this tab from:

- built-in knowledge about the processor
- target configuration information
- the description in the target vehicle.

For example, the first entry in the memory map shows Start. See Figure 6-3 on page 6-9) if the memory access rule is defined as Any. Colored icons are used to show the type of memory defined, see *Display colors* on page 6-8.

With an image loaded, the **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.

6.4.2 Memory map configuration

The memory map for a chosen processor is configured under the following headings:

Type	The type of memory page, for example Prog, I/O, Data. Where no such definition is given, the type is set to Any.														
Access	Defines the access rules for the memory: <table> <tr> <td>RAM</td><td>Memory is readable and writable.</td></tr> <tr> <td>ROM</td><td>Memory is read-only.</td></tr> <tr> <td>WOM</td><td>Memory is write-only.</td></tr> <tr> <td>NOM</td><td>No memory.</td></tr> <tr> <td>Auto</td><td>Memory is defined by the application currently loaded. If there is no application loaded, this shows NOM.</td></tr> <tr> <td>Prompt</td><td>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.</td></tr> <tr> <td>Flash</td><td>Memory is readable and, if a Flash programming method file (*.fme) is present, writable.</td></tr> </table>	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	Memory is readable and, if a Flash programming method file (*.fme) is present, writable.
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	Memory is readable and, if a Flash programming method file (*.fme) is present, writable.														
Start	The memory area is defined by the start address and the size. This defines the start address of the memory area.														
Size	Defines the size of the memory area.														
Access size	Defines the size of memory accesses.														
Filled	This shows if a range contains data loaded from an application program.														
Description	A text description of the purpose of the memory supplied as part of the automatic mapping. You can also supply this information yourself (see <i>Setting up a memory map</i> on page 6-5).														

To see details about a map entry, right-click on the chosen entry and select **Properties** from the context menu. This displays a text description of the type of memory defined at this location.

Display colors

When using the **Map** tab to view the memory map, RealView Debugger uses colored icons to make the display easier to read and to highlight the different memory definitions, shown in the example in Figure 6-3 on page 6-9.

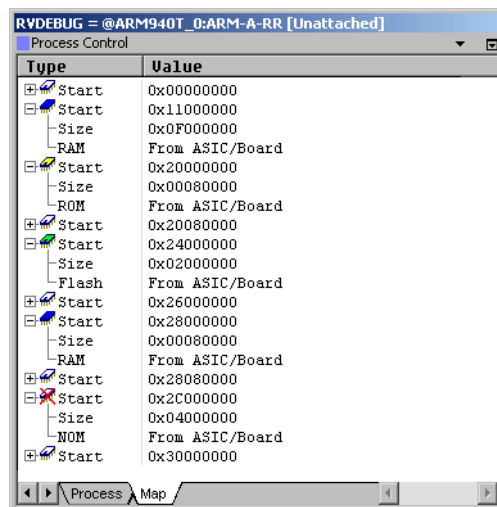


Figure 6-3 Colors in the memory map

In this example, memory has been mapped, using a Board/Chip definition file, to declare Flash. See the chapter that describes configuring custom targets, Chapter 14 *Configuring Custom Connections*, for details of configuring your target this way.

Colored icons enable you to identify the memory access defined:

- white (open) indicates one of the following:
 - no memory type is defined
 - you have set the memory to Auto, that is, auto-define as RAM when image loads into it
 - you have set the memory to Prompt, that is, prompt if image loads into it.
- blue indicates RAM
- yellow indicates ROM
- green indicates Flash memory known to RealView Debugger
- red indicates write-only memory (WOM)
- red cross indicates no accessible memory (NOM) is defined.

Also, see *Display colors* on page 6-8 for a description of the display colors in the Memory pane.

6.4.3 Using the Map tab context menu

The **Map** tab context menu enables you to add new mappings or to update existing ones. The options are:

Add or Copy Map Entry...

Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 6-1 on page 6-5, where you can create a new map entry based on an existing entry.

Edit Map Entry...

Displays the Add/Copy/Edit Memory Map dialog box, shown in Figure 6-1 on page 6-5, where you can edit a map entry.

Update Map based on Image

Updates the memory map based on details held in the loaded image.

Update Map based on Processor

Reads those registers that affect the 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 target configuration settings. Select this option to force RealView Debugger to read the registers and so update the memory map.

Save Map to Linker Command file...

Writes the current map state to a new or existing linker command file. This inserts or edits the MEMORY definitions in the linker command file, and enables the proper loading of an application based on actual memory settings. See *Generating linker command files for non-ARM targets* on page 6-14 for full details of how to generate this file for targets other than ARM targets.

Delete Map Entry

Deletes the map entry under the pointer. There is no undo.

Reset Map (Delete All)

Redefines the memory map to the initial state based on processor information, target configuration information, and processor registers. There is no undo.

Toggle Memory Mapping

Enables or disables memory mapping. When checked, memory mapping is enabled.

Properties...

Displays information about the selected memory map entry.

6.5 Editing map entries

You can edit any memory map entry except for those that have default mapping. To edit a memory map entry using the **Map** tab:

1. Right-click on the map entry in the display list to display the context menu.
2. Select the option **Edit Map Entry...** to display the Add/Copy/Edit Memory Map dialog shown in Figure 6-1 on page 6-5.
3. Use the Start Addr field to define the starting location for the mapping. This already contains the start address for the chosen block, shown in the **Map** tab.
4. Use the End field to define the block size for the mapping. By default, this specifies the size of the memory block to be defined. If you want to specify the end address, rather than the block size, unselect the check box **End is inclusive Length (vs. Addr)** and then enter the address in the End field, for example 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 resubmit the block size.

Entering a value of 0x0 remaps all memory from the starting address.

5. Highlight the access type in the display list, for example RAM.
6. Enter the memory type to be allocated, for example Any.
7. Enter a description of the new memory map settings, for example New test memory entry.
8. Click **OK** to confirm your new settings and to update the **Map** tab.

RealView Debugger displays a warning if you have entered any values incorrectly, for example a mismatch on start and end addresses. Correct these entries and click **OK**. When all entries are valid, the dialog box closes and RealView Debugger updates the **Map** tab.

6.5.1 Updating map entries based on registers

If you are connected to a debug target that uses register-controlled remapping, for example the ARM Integrator™/AP board, the **Map** tab also displays the effects of any changes made to these registers. In this case, right-click on the first entry and select **Update Map based on Processor** from the **Map** context menu (see *Using the Map tab context menu* on page 6-10) to update the display based on these memory-mapped registers.

6.6 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 stack base. Base of heap is then set to follow on directly from the end of the image data region.

You can create your own settings to specify the bottom of the stack address, the size of the stack, the bottom of the heap address, and the size of the heap. If you do not set these values manually, RealView Debugger uses default settings that are target-dependent. For ARM processors the default is `0x20000`.

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

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

To avoid this message, set permanent values for `top_of_memory`, stack base and limit, using the Connection Properties window. Configure your debug target and define these settings so that they are used whenever you connect with RealView Debugger. See your Getting Started Guide for further information.

You can set `top_of_memory`, and other ARM runtime controls specific to ARM processors, as part of a project definition. However, the available options depend on your target processor and target vehicle.

You can also set `top_of_memory`, stack, and heap values on a temporary basis, that is for the current session, using the `@top_of_memory` register. To do this select **Debug** → **Memory/Register Operations** → **Set Register...** to display the Interactive Register Setting dialog box, where the register contents can be changed.

6.7 Generating linker command files for non-ARM targets

The memory map, shown in the Process Control **Map** tab, 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. For details of how to set up such command options see the chapter describing customizing projects in the *RealView Developer Kit v2.1 Project Management User Guide*.

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 **Map** tab, or defined automatically from a loaded image, is allocated a name based on the characteristics of the memory.

The linker file format is processor-specific. If none is known, RealView Debugger uses a default format based on TI tools.

Example 6-1 on page 6-15 shows an example of a generated linker command file. The generated command file for your evaluation board might be different to this example.

Example 6-1

```

/* Linker Command file for the ARM processor */
/* This file was generated by RVDEBUG. You can edit everything
   outside the MEMORY block defined by RVDEBUG. Updates by
   RVDEBUG will only affect that block.*/
/* RVDEBUG: generated data block. Updated Wed Mar 19 17:23:41 2003
   Do not modify this block. Do not put MEMORY lines above
   this line, put below end of this block.*/
MEMORY
{
    /* Register @G_CM_CTRL has (masked) value 0000 */
    /* Register @G_CM_SDRAM has (masked) value 0000 */
    M_BootROM(R): org=0x0000, len=0x3FFFF /* external WaitStates=5 'Boot' */
    M_CM_Regs:    org=0x10000000, len=0xFFFFF /* external WaitStates=1 'CM
Registers' */
}
/* RVDEBUG: generated data above */

```

This example shows a combination of internal memory based on current register settings (@G_CM_CTRL and @G_CM_SDRAM) in addition to external memory as defined by the loading of a program.

The following notes apply to this automatic file generation process:

- If RealView Debugger creates the linker command file a comments section is inserted in the file reporting that it was generated by RealView Debugger, shown in Example 6-1.
- If a file already exists and is being updated by RealView Debugger, the comments section is not inserted. RealView Debugger then inserts the generated commands above the original user-generated commands.
- 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 7

Working with Debug Views

This chapter describes how to monitor your program during execution using panes and views in the RealView® Debugger Code window. It contains the following sections:

- *Working with registers* on page 7-2
- *Working with memory* on page 7-11
- *Working with the stack* on page 7-22
- *Using the call stack* on page 7-28
- *Working with watches* on page 7-32.

7.1 Working with registers

The Register pane displays the contents of processor registers and enables you to change those contents. Where appropriate, the pane shows registers using enumerations to make it easier to read the details, and enables you to enter new values in this format. The Register pane 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 your debug interfaces, see the appropriate documentation, for example:

- *RealView ICE and RealView Trace User Guide*
- *RealView ICE Micro Edition v1.1 User Guide*.

This section describes the options available when working with registers. It contains the following sections:

- *Displaying register contents*
- *Formatting options* on page 7-4
- *Changing register contents* on page 7-6
- *Understanding the register view* on page 7-7
- *Viewing debugger internals* on page 7-9
- *Defining new registers* on page 7-10
- *Interactive operations* on page 7-10.

7.1.1 Displaying register contents

To examine the contents of registers:

1. Select **View** → **Registers** to display the Register pane and bring the **Core** tab to the front.
2. Connect to your target and load an image, for example `dhrystone.axf`.
3. Select **Edit** → **Advanced** → **Show Line Numbers** to display line numbers. This is not necessary but might help you to follow the examples.
4. Click on the **Src** tab to view the source file `dhry_1.c`.
5. Set a default breakpoint by double-clicking on line 301.
6. Click **Run** to start execution.
7. 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.

8. The contents of the Register pane are updated to show the program status as the target stops, shown in Figure 7-1.

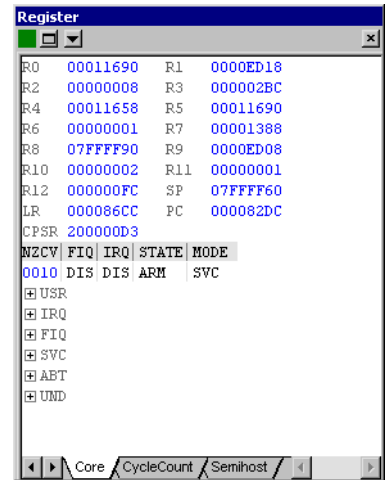


Figure 7-1 Register pane

9. Click on the **Core** tab to view base registers for your target processor.

The Register pane displays tabs appropriate to the target processor running your image and the target interface. Different target processors contain different registers and so the contents of this pane change depending on the target you are debugging.

————— Note —————

If your target processor is configured using .bcd files, additional tabs are also to be found to the right of the standard tabs shown in Figure 7-1.

For more details on the contents of this pane see:

- *Understanding the register view* on page 7-7 for information on registers.
- *Viewing debugger internals* on page 7-9 for information on debugger internals and statistics.

10. Click **High-level Step Into** to execute one instruction and then stop. Register values that have changed, since the last update, are displayed in dark blue.
11. Click **High-level Step Into** a few more times and examine the register values as they change.

12. Right-click on a changed register and select **View Memory At Value** to use the chosen value as the starting address for a memory display.

The first time you perform this operation in a debugging session, a new instance of the Memory pane is displayed as a floating pane. Subsequent **View Memory At Value** operations use this instance of the Memory pane to display the memory contents, even if you have docked that pane.

If you change the docked instance of the Memory pane to a different pane view (such as Process Control), and you later perform the **View Memory At Value** operation, the docked Memory pane is redisplayed to the right of the changed pane view (Process Control in this example).

This Memory pane is also used if you perform the view memory operations from the Watch pane or the File Editor pane.

13. Monitor changes in the Register pane as you step through your program.
14. Double-click on the red marker disc to clear the breakpoint at line 301.

Display colors

When using the Register pane to view 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.

7.1.2 Formatting options

You can change the way register values are displayed in the Register pane.

Global formatting of all registers

Click the **Pane** menu to select formatting options for all registers currently displayed. This applies formatting to all registers, except those registers that you have specifically formatted on an individual basis. For example, you might want to display contents as values rather than enumerations, or to display values in decimal format. This change applies to all tabs in the Register pane.

Use the option **Copy Pane** to take a snapshot of the top-level pane display. You can then copy this, for example into a text editor, so that you can compare registers and values.

For selected registers

You can change the display format for a single register while viewing other registers in the default format. Right-click on a chosen register, for example R3, to display the **Register** context menu.

This menu enables you to change the format of the chosen register, to view the properties, or to change the register contents, for example, select **Increment** to add one to the current value. You can also select from a list of previously used values to update the current contents.

The values of the FIQ, IRQ, STATE, and MODE fields of the CPSR register can be set using names that enumerate to numerical values. The options offered on this menu depend on the register currently under the mouse pointer. Position your pointer over the MODE field value of the CPSR register, which is set to SVC by default, and then right-click to display the status register context menu.

You can use this menu to change the register contents, or pick from a list of values appropriate to this register.

Select **Set Enumeration...** to display the selection box shown in Figure 7-2.

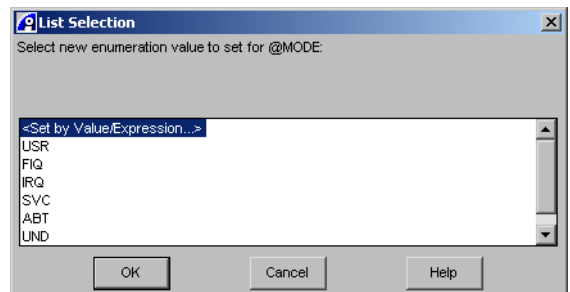


Figure 7-2 Setting enumeration values

This dialog enables you to select a new value for the chosen register or to use an expression to define the contents. For example, highlight the entry **<Set by Value/Expression...>**. Click **OK** to display a box where you can enter the value or choose from a drop-down list.

As you change a value in a CPSR register field, the CPSR register value is also updated.

7.1.3 Changing register contents

You can use in-place editing to change register contents in the Register pane. There are, however, other ways to set register values from the Code window:

- Set the contents of a register by pasting variables from other windows. For example, right-click on a value in the source-level view and select **Copy** from the context menu. Right-click in the register whose value you want to change and select **Paste Value** from the context menu.
- If you want to use a favorite data value that you have saved, right-click on a register and select **Set from Favorites** from the context menu to display a Favorites Chooser/Editor dialog box (see *Using the Favorites Chooser/Editor dialog box* on page 9-29).
- Highlight a value in the **Src** tab in the File Editor pane and then use drag-and-drop to copy the expression into the contents of a chosen register in the Register pane. See *Dragging and dropping expressions into register contents* for an example.
- Select **Debug** → **Memory/Register Operations** → **Set Register...** from the Code window main menu to display the Interactive Register Setting dialog box.

Any register contents you change are displayed in blue.

————— **Note** —————

You can also set breakpoints on memory mapped registers but not on core registers, because breakpoints are set on addresses and core registers do not have addresses.

Dragging and dropping expressions into register contents

You can drag and drop an expression from a source file into the contents of a register function name. The following example shows how you can set the PC register to the address of a function:

1. Load the `dhrystone.axf` image.
2. Enable line numbering and locate line 149 in the `dhry_1.c` source.
3. Double-click on the `Proc_5` function name.
4. Click on the function name, and drag it to the PC register.

The address of the `Proc_5` function is loaded into the PC register, and the source level view changes to show the code for the `Proc_5` function. The **Dsm** tab is also updated.

Note

If the function code is in a different source file, and that source file is not open, RealView Debugger cannot determine the context, and informs you of this in the **Src** tab. However, the **Dsm** tab is updated.

Updating register contents

The Register pane updates automatically when register contents change. This is set by default, as indicated by the checked option **Automatic Update** on the **Pane** menu.

If you change this default option, you must update the pane display manually. Select **Update Window** from the Register pane context menu to update the view. RealView Debugger updates the pane contents and displays changed values in blue for improved readability.

7.1.4 Understanding the register view

ARM processors support up to seven processor modes depending on the architecture version, for example User (USR), Supervisor (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, IRQ, and FIQ for an ARM7TDMI® core. Click on the plus, or minus, sign to expand, or collapse, the view (shown in Figure 7-3 on page 7-8).

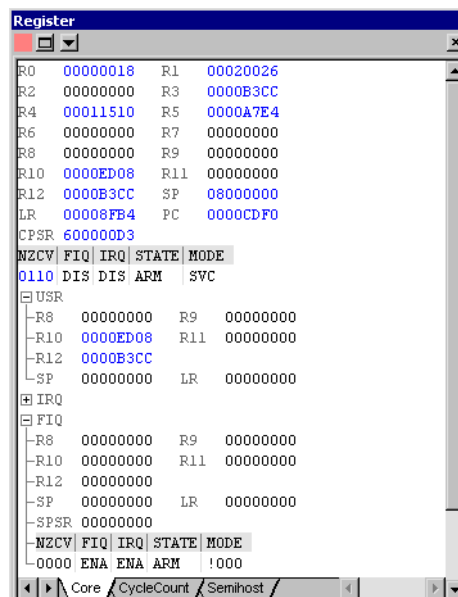


Figure 7-3 Viewing registers

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 SPSR. 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.

For full details on LDM and STM commands see the appropriate documentation, for example *RealView Developer Kit Assembler Guide*.

7.1.5 Viewing debugger internals

Depending on your debug target, debugger internals appear in tabs in the Register pane:

- *Viewing internal variables*
- *Viewing semihosting controls* on page 7-10.

Viewing internal variables

RealView Debugger uses internal variables like any other program. These are set up as part of the RVI/RVI-ME or the .bcd file configuration. See *Creating a new .bcd file* on page 13-15 for further information. The variables it uses depend on your debug target. An example of the **Debug** tab for RVI is shown in Figure 7-4.

Note

It is strongly recommended that you do not change the semihosting ARM and Thumb® SWI numbers. If you do, you must:

- change all the code in your system, including library code, to use the new SWI number
- change the ARM_SWI and Thumb_SWI settings to use the new SWI numbers.

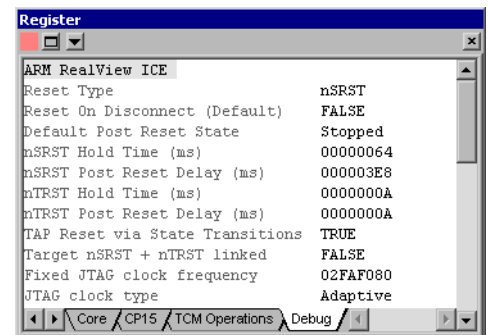


Figure 7-4 Viewing internal variables (RVI)

For information on what is shown for different target interfaces, see the appropriate documentation:

- *RealView ICE and RealView Trace User Guide*
- *RealView ICE Micro Edition v1.1 User Guide.*

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 disk I/O.

The debugger contains BCD files that describe MCUs and corresponding development boards. The actual files vary between versions of RVDK. Board files may have semihosting and vector catch on by default. This section describes how to turn off these features when running code from Flash memory.

Note

RealView Debugger does not currently support the use of channel viewers.

Standard semihosting behavior

If there is ROM or FLASH at the SWI Vector, then the use of semihosting requires a hardware breakpoint on certain cores, such as 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.

7.1.6 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 Register pane. However, you can define new ASIC registers using the Advanced_Information blocks in your target configuration settings. When configured, user-defined registers can be displayed in the Register pane in the same way as standard registers.

7.1.7 Interactive operations

For full details on using interactive operations on register contents using the **Debug** menu, see Chapter 8 *Reading and Writing Memory, Registers and Flash*.

7.2 Working with memory

The Memory pane displays the contents of memory and enables you to change those contents. On first opening, the pane is empty, because no starting address has been specified. If a starting address is entered, values are updated to correspond to the current program status each time your program stops.

Note

RealView Debugger can only connect to ARM® architecture-based processors.

This section describes the options available when working with memory displays and contains the following sections:

- *Displaying memory contents*
- *Formatting options* on page 7-13
- *Operating on memory contents* on page 7-17
- *Data width on memory accesses* on page 7-19
- *Changing memory contents* on page 7-19
- *Interactive operations* on page 7-21.

7.2.1 Displaying memory contents

To examine the contents of memory:

1. Select **Target** → **Reload Image to Target** to reload the image dhrystone.axf.
You can also reload an image using the **Reload Image** button on the Image toolbar.
2. Click on the **Src** tab to view the source file dhyr_1.c.
3. Set a default breakpoint by double-clicking on line 301.
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. If no Memory pane is visible, select **View** → **Memory** to display the Memory pane, shown in Figure 7-5 on page 7-12.
You can set start addresses using in-place editing or using the context menu.

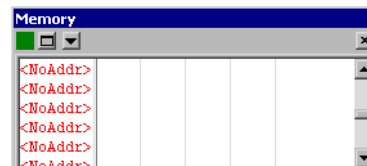


Figure 7-5 Memory pane

7. Right-click in the address column in the Memory pane to display the **Address** context menu.
8. Select **Set Start Address...** to display the dialog box shown in Figure 7-6.

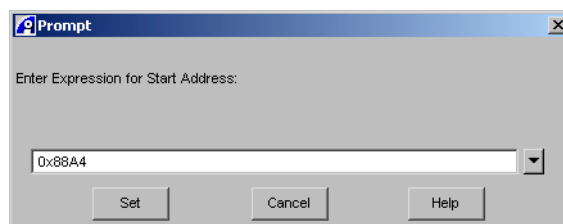


Figure 7-6 Memory start address selection box

You specify the start address by giving an address in hexadecimal or by giving a C/C++ expression that RealView Debugger computes to obtain the starting address. You can use any valid expression using constants and symbols.

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 (see *Using the Favorites Chooser/Editor dialog box* on page 9-29) where you can store a memory address for re-use in this, or future, debugging sessions.

In this example, memory addresses of interest are in the region of 000088A0 so set the start address to examine memory from this location.

Numbers entered here must start with a zero. This means that RealView Debugger can distinguish these entries from valid variable names.

9. Enter the required location, for example 0x00088A4, and then click **Set** to update the Memory pane.

The memory display is arranged in columns. The left-most column shows the memory address. The memory contents are shown in the other columns. The number of columns displayed varies depending on the size of the pane. Color coding is used to distinguish the type of memory being displayed, see *Display colors* on page 7-16 for details.

10. Monitor changes in the memory display as you step through your program.

11. Double-click on the red marker disc to clear the breakpoint at line 301.

7.2.2 Formatting options

You can change the way memory contents are displayed, and set the start address, using the **Pane** menu:

Copy If you have selected memory contents, use this option to copy the values to the clipboard ready to paste.

Update Window

If you have unselected the option **Automatic Update**, you can use this option to update the memory display manually. You can update the display using this option at any time when execution is stopped. This enables you to catch any memory updates made externally.

Set Start Address...

Select this option to enter a C/C++ expression to compute the start address. This displays the selection box shown in Figure 7-6 on page 7-12 with the current expression already displayed. Change the address and click **Set** to specify the start address for the memory view.

Previous Start Address

Uses a previous start address for displaying the contents of memory.

The history list holds up to 16 previous start addresses added when:

- you enter a new start address or expression
- the current expression is recomputed to generate a new start address
- the start address is set from the **Address** context menu.

Re-evaluate Start Address

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression and, where necessary, start at the new location. Where you have used a fixed value to specify the start address, select this option to update only the pane contents.

Set Number of Columns to show...

When the Memory pane first opens, the number of columns you can see depends on the size of the pane and is chosen so as to show an even number of bytes. Use this option to change the number of columns visible in the display. Use the selection box to show up to 32 columns in a single

window. This number does not include the column used when the ASCII display option is selected, see the description of **Show ASCII** in *Data sizes*.

The default setting is 0 to configure the number of columns to fit the window size.

Automatic Update

Updates the memory display automatically, that is when:

- you change memory from anywhere in RealView Debugger
- program execution stops.

This is the default.

Re-evaluate Start Address on Update

Where you have used a C/C++ expression to compute the start address, select this option to recompute the expression when the pane contents are updated (see the description of **Automatic Update**), and start at the new computed location where necessary.

Size Displays a submenu where you can select various data sizes (see *Data sizes*).

Format

Displays a submenu where you can select various data formats (see *Data formats* on page 7-15).

Data sizes

The **Pane** menu contains a **Size** submenu to define how data values are displayed in the Memory pane. The display size used for viewing memory contents varies depending on the data types supported by your target processor:

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

This 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 pane.

Show ASCII Adds another column to the Memory pane, 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 (.).

Data formats

The **Pane** menu contains a **Format** submenu to define how data values are displayed in the Memory pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

Minimum Access Size

Displays memory contents in the format specified as the minimum memory access size for the target. This is the default.

Bytes (8 bits) Each column displays 8 bits of data.

Half Words (16 bits)

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

Words (32 bits)

Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

Double Words (64 bits)

Each column displays 64 bits of data.

Fixed (word size)

Enables you to use fixed point format for displaying numeric values, that is based on the natural size for the debug target processor. The default format is unsigned and one less than the number of bits in the value.

Fixed... Displays a selection box that enables you to specify a fixed point format to display numeric values. The value entered here becomes the default display format for the pane.

Floats (32 bits)

Displays values in floating point IEEE format, occupying four bytes, for example:

2.5579302e-041

Doubles (64 bits)

Displays values in floating point IEEE format, occupying eight bytes, for example:

4.71983561663e+164

Display colors

When using the Memory pane to view memory contents, RealView Debugger uses color to make the display easier to read and to highlight significant events:

- Black specifies RAM or memory that can be modified.
- Blue shows those contents that have changed since the last update. Light blue indicates a previous update.
- Yellow indicates the contents of ROM.
- Green indicates Flash memory known to RealView Debugger. Otherwise the values are displayed in yellow, indicating ROM.

Note

If the Flash memory locations are shown with a yellow background in the Memory pane, 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 one of:
 - no memory is defined 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 that there has been a failure in performing the memory operation. Double-click, with the right mouse button, at this location to get an explanation of the problem.

7.2.3 Operating on memory contents

You can perform different operations on the memory displayed in the Memory pane using the context menus. The menu shown, and the options available, depend on the type of memory under the cursor when you right-click and on the valid licenses that you have. The color-coded display helps you to identify the memory type. For a description of the context menus, see:

- *Address context menu*
- *Memory value context menu* on page 7-18.

Note

If you right-click on a memory cell to access a context menu, any change you specify is made to the cell under the cursor. This is independent of any highlighted cells in the view.

If you double-click on an address in the address column of the Memory pane, an edit field is displayed. You can enter a specific address, or a symbol that specifies an address, in this field. For example, to display memory at the address pointed to by the SP register, enter @SP.

Address context menu

Right-click on a memory address to display the **Address** context menu that provides options to set the start address:

Update

Updates the display in the Memory pane.

Set Start Address...

Enables you to specify the starting address for the display of memory contents.

Previous Start Address

Enables you to use the previous starting address for the display of memory contents.

Re-evaluate Start Address

Where you have used a C expression to compute the start address, select this option to recompute the expression and, where necessary, start the display at the new location.

Memory value context menu

Right-click on a memory cell (or byte) shown as black or green, that is where the type is ROM, Flash, or modifiable, to display the **Memory value** context menu. This context menu contains the following options:

Update

Updates the display in the Memory pane.

Set Start Address from Content

Enables you to use the cell contents as the starting address for the display of memory contents. This option is enabled when the cell contains a scalar the size of an address or pointer.

Show Symbol from Content

RealView Debugger looks up the address held in the cell and displays any symbol at that address.

Show Symbol at Address

Displays any symbol at the address contained in the cell, and not the contents.

Set to 0 Enables you to set the current memory cell to zero.

Increment Enables you to add 1 to the contents of the memory cell.

Decrement Enables you to subtract 1 from the contents of the memory cell.

Set Value... Displays a prompt where you can enter a new value for the memory cell. This new value is then entered and the memory display is updated.
A memory cell can also be changed using in-place editing.

Set Memory Interactive...

Enables you to use memory interactive operations available in RealView Debugger.

Fill Memory with Pattern...

Enables you to fill memory starting at this location.

Set Break At...

Displays the Set Address/Data Break/Tracepoint dialog box where you can specify a breakpoint on the current memory cell. The type of breakpoint offered depends on the type of memory at the chosen location. For example, if the memory is defined as ROM, RealView Debugger offers a hardware breakpoint first.

Set Tracepoint...

Enables you to set tracepoints based on the current memory view.

Right-click on a memory cell, or byte, that is yellow, that is where the type is ROM, to see the ROM-specific context menu that offers a subset of these options.

Memory errors

Where a memory cell contains !!!! (colored red), this shows that there has been an error in the memory operation. Right-click to display a menu with a single option:

Show Error Code...

Select this to display the error code returned from the debug target when the memory operation failed.

7.2.4 Data width on memory accesses

By default, RealView Debugger makes word-size accesses when reading or writing memory. You can change this in your memory map or by specifying the `Memory_block` settings in the `Advanced_Information` block in your target configuration file.

7.2.5 Changing memory contents

To change memory contents:

1. Select **Target** → **Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
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 **Pane** menu, in the Memory pane, to set the start address to `0x89A2` and display the ASCII values, shown in Figure 7-7 on page 7-20.

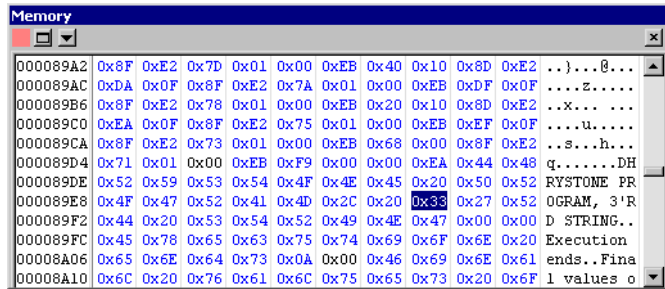


Figure 7-7 Example memory display

The memory locations 000089EF-000089F2 contain the four hexadecimal values 0x33, 0x27, 0x52, and 0x44 corresponding to the string 3'R D.

Right-click in the value at 000089ef, that is 0x33, to display the **Memory value** context menu (see *Operating on memory contents* on page 7-17).

This enables you to change the contents at the specified location. Select the option **Set Value...** from the context menu to display the selection box, shown in Figure 7-6 on page 7-12, where you can enter the new value.

Enter the required hexadecimal value 0x4E, or enter 'N', and click **Set** to update the memory display. You can use uppercase or lowercase to enter the new value.

The new value is displayed in blue and the ASCII value changes from 3 to N.

You can also use the drop-down arrow to select 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 data value for re-use in this, or future, debugging sessions.

7. Change the value at location 000089F0 from 0x27 to 0x6F (lowercase o).
8. Data values can be entered in a format that is different from the display format. Right-click in the location 000089F1 (0x52), for example, enter the decimal value 32 and then click **Set**. The memory pane is updated with the new value, a space, displayed in the chosen display format.
9. You can also use in-place editing to change memory contents. Double-click in the value 0x44 (D) at location 000089F2 and change it to 0x32. Press Enter to confirm the new value.
If you press Escape then any changes you made in the highlighted field are ignored.
10. View the changed values in the memory display. Each new value is displayed in blue as the pane contents are updated.

11. View the changes you have made in the messages displayed when your program completes. The string “3’RD” has been replaced by “No 2”.
12. Double-click on the red marker disc to clear the breakpoint at line 301.

7.2.6 Interactive operations

You can also perform operations on memory contents including saving memory to a file, and reloading, and filling memory. See Chapter 8 *Reading and Writing Memory, Registers and Flash* for details.

7.3 Working with the stack

The stack, or run-time stack, is an area of memory used to store function return information and local variables.

Executing a function sets up the stack. As the new function is called, a record is created on the stack including traceback details, and 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 recovered automatically and can then be used for the next function call.

In a typical memory-managed ARM processor, the memory model comprises:

- 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.

The *Stack Pointer* (SP) points to the bottom of the stack.

Note

Modifying a value in the stack might cause the application program to perform incorrectly or even to abort operation completely.

RealView Debugger can provide the calling sequence of any functions that are still in the execution path because their calling addresses are still on the stack. However, when the function is off the stack, it is lost to RealView Debugger. Similarly, if the stack contains a function for which there is no debug information, RealView Debugger might not be able to trace back past it.

This section describes ways of working with the stack:

- *Using the Stack pane* on page 7-23
- *Formatting options* on page 7-24
- *Operating on stack contents* on page 7-25
- *Context controls* on page 7-26
- *Setting a breakpoint* on page 7-26
- *Interactive operations* on page 7-27.

7.3.1 Using the Stack pane

The Stack pane enables you to monitor the contents of the stack as raw memory, and to make changes to those settings. This might be especially useful for assembly language programmers. The Stack pane shows the contents of the stack at the SP register which is always kept at the top-left of the display area. Use this pane to view changes as they happen in the stack.

The Stack pane enables you to 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.

To view the Stack pane:

1. Select **Target** → **Reload Image to Target** to reload the image dhrystone.axf.
2. Select **View** → **Stack** to view the Stack pane.
3. Click on the **Src** tab to view the source file dhry_1.c.
4. Set a default breakpoint by double-clicking 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 red box marks the location of the PC when execution stops.

7. View the updated Stack pane, shown in Figure 7-8.

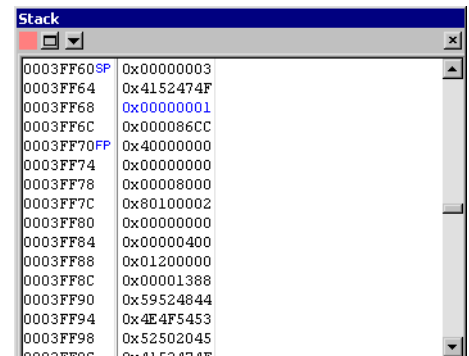


Figure 7-8 Viewing the stack

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.

8. Monitor changes in the Stack pane as you step through your program, for example by clicking **Hi-level Step Into**.
9. Double-click on the red marker disc to clear the breakpoint at line 301.

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, a D is appended to show that this is a data address. You must include this letter when specifying such an address as the starting address.

Value The right column displays the contents of the addresses in the stack.

As with the Memory pane, the memory display in the Stack pane is color-coded for easy viewing and to enable you to monitor changes.

7.3.2 Formatting options

You can change the way stack contents are displayed, and set the start address, using the **Pane** menu. This menu provides options to manage the display of stack contents during your debugging session, change the display format, and extract data from the pane for use in other panes or windows. Highlight an entry in the display and then choose from the list of available options:

Copy Copies the chosen entry from the list to the clipboard.

Previous Start Address

Enables you to use the previous starting address for the display of memory contents.

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 pane.

Show ASCII Adds another column to the Stack pane, 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 (.).

Data formats

The **Pane** menu contains an extended panel to define how data values are displayed in the Stack pane. The display format used for viewing memory contents varies depending on the data types supported by your target processor:

Bytes (8 bits) Each column displays 8 bits of data.

Half Words (16 bits)

Each column displays 16 bits of data. Where your debug target is an ARM processor halfwords are aligned on 2-byte boundaries.

Words (32 bits)

Each column displays 32 bits of data. Where your debug target is an ARM processor long words are aligned on 4-byte boundaries.

Double Words (64 bits)

Each column displays 64 bits of data.

7.3.3 Operating on stack contents

You can perform operations on the memory displayed in the Stack pane using the memory contents context menus, as described in *Operating on memory contents* on page 7-17.

Changing the stack pointer

As you step through your code, the default stack pointer is used, shown in Figure 7-8 on page 7-23. You can specify an expression or a register to use as the stack pointer from the Stack pane:

1. Right-click on a stack address or the Stack pane background to display the **Stack** context menu.

Note

If you right-click on a stack value, the **Stack Value** context menu is displayed.

2. Select **Set Start Address...** to display the address prompt box.
3. Enter an expression or a register, for example @R9, as the new stack pointer.
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 (see *Using the Favorites Chooser/Editor dialog box* on page 9-29) where you can store a memory address for re-use in this, or future, debugging sessions.
4. Click **Set** to confirm your choice and close the address prompt box.

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

If you enter a blank expression, or remove the existing expression, in the address prompt box, RealView Debugger reverts to using the default stack pointer register. In this example, this was R13 shown in Figure 7-1 on page 7-3.

7.3.4 Context controls

There are two **Context** controls available from the Code window main menu:



Stack up Moves up one stack level from the current scope location giving access to all local variables at that location. A stack level is determined by each calling function.



Stack down Moves down one stack level from the current scope location giving access to all local variables at that location. A stack level is determined by each calling function.

You must use the **Stack up** control first, because the context is as far down the stack as possible.

7.3.5 Setting a breakpoint

To set a breakpoint in the Stack pane:

1. Right-click on the required stack value to display the **Stack Value** context menu.
2. Select **Set Break At...** from the **Stack Value** context menu.
3. Complete the entries in the Set Address/Data Breakpoint dialog box.
4. Click **OK** to close the dialog box and set the 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. Do not, therefore, use such a breakpoint where execution runs past the function return call.

Unlike the Watch pane, the Stack pane 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.

7.3.6 Interactive operations

You can also perform other operations on memory contents using the Stack pane. See Chapter 8 *Reading and Writing Memory, Registers and Flash* for details.

7.4 Using the call stack

Processors maintain a call stack for each processor in your debug target. 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.

RealView Debugger includes features that enable you to monitor variables and access traceback as your debugging session develops:

- *Using the Stack pane*
- *Using the Call Stack pane.*

7.4.1 Using the Stack pane

The Stack pane enables you to monitor activity on the stack during program execution by giving access to the stack as raw memory. See *Working with the stack* on page 7-22 for full details.

7.4.2 Using the Call Stack pane

Use the Call Stack pane to follow the flow of your application through the hierarchical structure by examining the current status of functions and variables. This enables you to see the path that leads from the main entry point to the currently executing function at the top of the stack. You can also use the Call Stack pane to set breakpoints.

The Call Stack pane shows the:

- name of the function
- line number in the source file from which the function was called
- parameters to the function.

To use traceback:

1. Select **Target** → **Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set an unconditional software breakpoint by double-clicking on line 301.
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. Select **View** → **Call Stack** to view the Call Stack pane, shown in Figure 7-9.

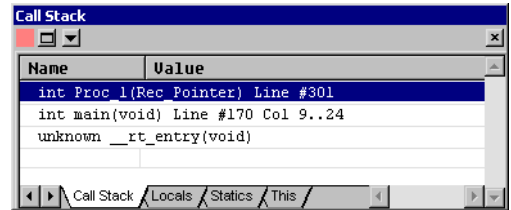


Figure 7-9 Multistatement details in the Call Stack pane

7. Continue to step through your program. If the current line is a multistatement line, the Call Stack pane shows the information in the form of line and column details, shown in Figure 7-9.
8. Monitor changes in the Call Stack pane as you step through your program.
9. Double-click on the red marker disc to clear the breakpoint at line 301.

Tabs in the Call Stack pane

The Call Stack pane contains the tabs:

- | | |
|-------------------|--|
| Call Stack | Displays details of the functions currently on the stack, shown in Figure 7-9. |
| Locals | Displays a list of the variables that are local to the current function. |
| Statics | Displays a list of static variables local to the current module. |
| This | In C++ the this pointer locates the object for which the member function was called. It is C++ specific. |

Using the Pane menu

Click on the **Pane** menu button to display the **Pane** menu that contains options to:

- manage variables
- change the display format
- change how contents are updated
- extract data from the Call Stack pane for use in other panes or windows.

Click on the **Call Stack** tab, highlight an entry in the functions list and display the **Pane** menu options:

Copy Copy the chosen entry, where enabled.

Timed Update when Running

Not available in this release.

Timed Update Period...

Not available in this release.

Update Window

Updates the contents of the Call Stack pane. Use this when **Automatic Update** is disabled.

Automatic Update

Refreshes the Call Stack pane as soon as a watchpoint is triggered and execution stops. This is enabled by default.

Show char* and char[] as strings

Displays local variables 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.

Properties... Displays a text description of the item under the cursor.

7.4.3 Using context menus

The **Call Stack** pane contains several context menus depending on which entry is selected and the type of variable under the cursor.

Using the Function menu

Right-click on a function in the **Call Stack** tab to display the **Function** context menu:

Scope To Scopes to the chosen function.

Break at... Sets a breakpoint at the address defined by the chosen function, if this is permitted.

BreakIf... Displays the Breakpoint type selection box.

- Go to** This continues execution until the specified point in the stack is reached.
- Properties...** Displays a text description of the item under the cursor.

Using the Variable menu

Right-click on a chosen entry Name or Value in the **Locals** tab to display the **Variable** menu. This context menu provides options that operate on selected variables only:

- Update** Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the **Pane** menu.
- Format...** Displays a selection box where you can highlight the required format for the expression from the list of available formats.
- If a variable contains multibyte characters, you can choose to view it using ASCII, UTF-8, or Locale encoding.

7.4.4 Stack controls

When working with the Call Stack pane there are two **Context** controls available from the Debug toolbar:

- Stack up
- Stack down.

See *Context controls* on page 7-26 for full details.

7.5 Working with watches

Use watches to monitor variables or to evaluate expressions during your debugging session:

- *Setting watches in source-level view*
- *Working with the Watch pane*
- *Managing watches on page 7-34*
- *Saving watches as favorites on page 7-37.*

7.5.1 Setting watches in source-level view

To set a watch:

1. Select **Target** → **Reload Image to Target** to reload the image dhrystone.axf.
2. Click on the **Src** tab to view the source file dhry_1.c.
3. Right-click on a variable name, for example Run_Index at line 146, and select **Watch** from the **Source Variable Name** menu.
4. Set a second watch, for example on the variable Enum_Loc at line 155.

If you are working in source-level view and the Watch pane is visible, you can set watches in other ways:

- use the option **Enter New Expression...** from the **Pane** menu in the Watch pane
- drag-and-drop the variable to watch
- select a variable to watch, copy it, and paste it into the Watch pane.

7.5.2 Working with the Watch pane

The Watch pane enables you to view expressions and their current values. You can use the Watch pane to create breakpoints, or to change existing watched values.

Displaying the pane

Select **View** → **Watch** to view the Watch pane, shown in Figure 7-10 on page 7-34.

The Watch pane contains a series of tabs. The **Watch1** tab is selected by default. You can set expressions on different tabs to make it easier to manage what is being watched during your debugging session. Click on the required tab to select it.

Expressions are listed in the order they were 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.

Using the Pane menu

Highlight an entry in the watched variables list and click the **Pane** menu button to display the **Pane** menu. This contains:

Copy Select this option to copy the chosen value from the list to the clipboard. From here it can be copied into another tab in the Watch pane, or into another pane or window.

Update Window

Updates the contents of the Watch pane. Use this when **Timed Update when Running** and **Automatic Update** have been disabled. Select this to update the current tab.

Enter New Expression...

Displays a prompt box where you can enter the expression to be watched. This displays the name and current value in the current tab. Click the required tab before selecting this option.

Automatic Update

Refreshes the Watch pane as soon as execution stops. This is enabled by default.

Recompute Expression with same Context

If you watch an expression, the result is evaluated based on the current context. Select this option to recompute expressions using the context when set.

Show char* and char[] as strings

Displays values as strings where appropriate. 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.

Properties... Displays a text description of the item under the cursor.

Viewing watches

As you set watches, expressions are added to the Watch pane, shown in Figure 7-10 on page 7-34.

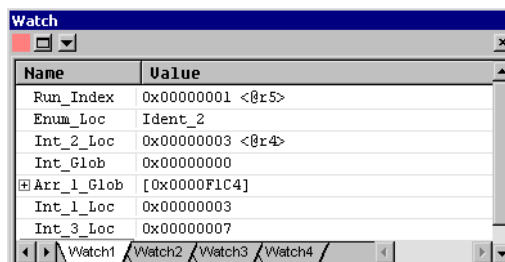


Figure 7-10 Watch pane with watches

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 on the boundary marker to make the details easier to read.

To delete an expression, highlight it and press Delete. There is no undo for this operation but saving the watches in your personal Favorites List enables you to reinstate any deleted entry.

7.5.3 Managing watches

With a watch set, you might want to change the expression, change the display format used, or edit it directly to control how it is monitored. The Watch pane enables you to carry out these operations and gives access to context menus where editing options are available.

Figure 7-11 shows an example Watch pane, with several expressions already set.

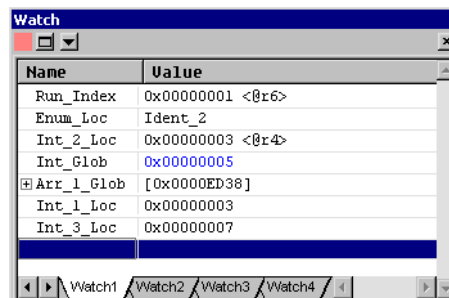


Figure 7-11 Watches in the Watch pane

One watched value is an array, shown by the plus sign appended to the variable name. Click on 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.

You can access context menus, inside the Watch pane, to control watch options and edit watches directly, see:

- *Using the Name menu*
- *Using the Value menu* on page 7-36
- *Using the pane context menu* on page 7-37
- *Editing watches* on page 7-37.

Using the Name menu

If you have set up expressions, right-click on a chosen entry Name to display the **Name** menu. This context menu provides options that operate on selected entries only:

- Update** Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the **Pane** menu.
- Format...** Displays a selection box to specify the format for the watched expression. Highlight the required format from the list of available formats. You can also cast top level expressions, including casting to array, for example `char*` and `char[12]`.
Click **OK** to confirm your choice. This closes the selection box and the new format is applied to the chosen expression.
If a variable contains multibyte characters, you can choose to view it using ASCII, UTF-8, or Locale encoding.
- Break at...** If enabled, select this option to set a breakpoint at this location.
- BreakIf...** If enabled, select this option to set a conditional breakpoint.
- Add from Favorites...**
Displays the Favorites Chooser/Editor dialog box where data values saved in your personal favorites can be added to the Watch pane (see *Using the Favorites Chooser/Editor dialog box* on page 9-29 and *Saving watches as favorites* on page 7-37 for details).

View Memory at Value

Select this option to use the chosen value as the starting address for a memory display.

View Memory at Address

Select this option to use the address of the chosen item as the starting address for a memory display.

Properties... Displays a text description of the item under the cursor.

Note

The first time you perform the **View Memory at Value** or **View Memory at Address** operations in a debugging session, a new instance of the Memory pane is displayed as a floating pane. Subsequent view memory operations use this instance of the Memory pane to display the memory contents, even if you have docked that pane.

If you change the docked instance of the Memory pane to a different pane view (such as Process Control), and you later perform the **View Memory At Value** operation, the docked Memory pane is redisplayed to the right of the changed pane view (Process Control in this example).

This Memory pane is also used if you perform the view memory operations from the Register pane or the File Editor pane.

Using the Value menu

With a watch set, you can right-click on a chosen entry Value to display the **Value** menu. This context menu provides options that operate on selected watches only:

- | | |
|------------------|---|
| Update | Updates the displayed value for the chosen expression. This is applied in combination with any update options you set from the Pane menu. |
| Format... | <p>Displays a selection box where you can highlight the required format for the expression from the list of available formats.</p> <p>If a variable contains multibyte characters, you can choose to view it using ASCII, UTF-8, or Local encoding.</p> |
| Set to 0 | Sets the value of the chosen expression to zero, where allowed. An error message is displayed if you try to set a value to zero when not permitted. |
| Increment | Adds 1 to the value of the chosen expression. An error message is displayed if you try to increment a value when not permitted. |
| Decrement | Subtracts 1 from the value of the chosen expression. An error message is displayed if you try to decrement a value when not permitted. |

Set from Favorites...

Displays the Favorites Chooser/Editor dialog box where data values saved in your personal Favorites List can be inserted into the specified location (see *Using the Favorites Chooser/Editor dialog box* on page 9-29 and *Saving watches as favorites* for details).

Recent expressions

The rest of this menu contains a list of recently-used variables and data values. You can re-use entries from this list as required.

Using the pane context menu

If you right-click anywhere inside an empty entry in the Watch pane, a short context menu is displayed. This provides the options:

Update All Updates the details for all the expressions currently displayed in the Watch pane.

Add from Favorites...

Displays the Favorites Chooser/Editor dialog box where expressions saved in your personal Favorites List can be added to the Watch pane (see *Using the Favorites Chooser/Editor dialog box* on page 9-29 and *Saving watches as favorites* for details).

Editing watches

You can use in-place editing to change expressions in the Watch pane, and to add new ones:

1. Double-click in the name you want to change, or press Enter if the item 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. Press Enter to store the new name.

If you press Escape then any changes you made in the highlighted field are ignored.

7.5.4 Saving watches as favorites

You can create watches and then add them directly to your personal Favorites List or you can add watches that you have been using in the current debugging session. This section explains the steps to follow to do both.

Creating a watch favorite

To create a Watch favorite, right-click inside a blank entry of the Watch pane to display the **Name** menu, and then select **Add from Favorites....** This displays the Favorites Chooser/Editor dialog box. If this is the first time you have used watches in RealView Debugger, the display list is empty.

To create a new watch and add it to your Favorites List:

1. Click **New** to display the New/Edit Favorite dialog box shown in Figure 7-12.

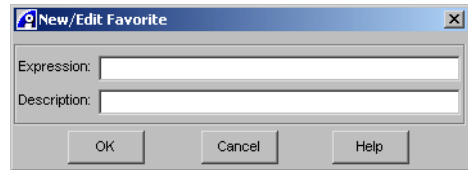


Figure 7-12 New/Edit Favorite dialog box

2. Enter the expression to be watched, for example `Ptr_Comp`.
3. Enter a short text description to help you to identify the watch for future use, for example `my watch favorite`.

This is optional.

4. 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 watch shown in the display list (shown in Figure 7-13 on page 7-39). See *Using the Favorites Chooser/Editor dialog box* on page 9-29 for details.

Duplicate entries are not permitted in the Favorites List.

Saving existing watches as favorites

With several watches already set, RealView Debugger lets you choose which to add to your Favorites List so that they are available for re-use in future debugging sessions or with other target configurations of your application program.

To add existing watches to your Favorites List:

1. Highlight an expression in the Watch pane that you want to add to your Favorites List.
2. Right-click on the Name and select **Add from Favorites...** from the **Name** menu. This displays the Favorites Chooser/Editor, shown in Figure 7-13 on page 7-39.

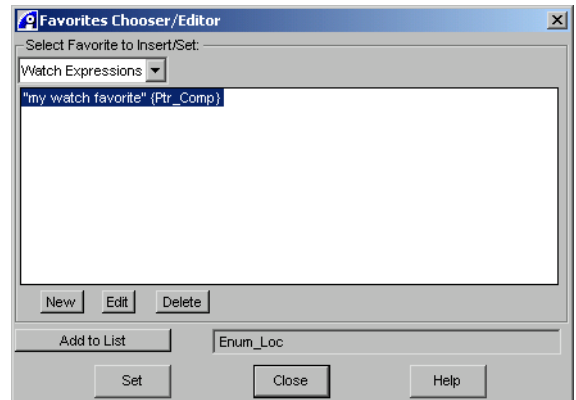


Figure 7-13 Existing watches in the Favorites Chooser/Editor

The display list shows any watches already saved in your Favorites List. The data field now shows the chosen expression.

3. Click **Add to List** to add the specified expression to your Favorites List. This displays the New/Edit Favorite dialog box shown in Figure 7-14.

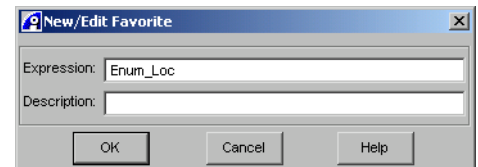


Figure 7-14 Adding a new favorite

The Expression field contains the chosen watch and you can enter a short text description to help you identify the watch favorite. This is optional.

4. Click **OK** to confirm the watch details and close the dialog box.

The Favorites Chooser/Editor dialog box is displayed showing the new watch in the display list. Because this watch is already set, click **Close** to close the dialog box. If required, set another watch from your Favorites List before closing the dialog box.

The edited Favorites List is saved to your `exphist.sav` file when you close RealView Debugger.

Saving data values as favorites

With several watches already set, you can right-click on the Value for a specified expression and display the **Value** menu. This context menu includes the option **Set from Favorites...** to specify a data value to be set.

Select this option to display the Favorites Chooser/Editor dialog box where you can:

- save an existing data value as an entry in your Favorites List so that it can be re-used later in this debugging session
- take a data value already saved and use it to set the starting value for the specified watch.

Chapter 8

Reading and Writing Memory, Registers and Flash

RealView® Debugger includes options that enable you to work with registers and memory interactively during your debugging session. This chapter describes these options. It contains the following sections:

- *About interactive operations* on page 8-2
- *Using the Memory/Register Operations menu* on page 8-3
- *Accessing interactive operations in other ways* on page 8-4
- *Working with Flash* on page 8-5
- *Examples of interactive operations* on page 8-10.

8.1 About interactive operations

Use interactive operations to:

- set memory and registers
- patch assembly code (where supported by your debug target)
- read a file to memory
- write memory to a file
- verify memory against a file
- fill memory with a pattern of your choosing
- control Flash memory.

You can access all these features directly from the **Debug** menu. Selected operations are also available when you are working in panes. These are described in:

- *Using the Memory/Register Operations menu* on page 8-3
- *Accessing interactive operations in other ways* on page 8-4
- *Working with Flash* on page 8-5.

The last section in this chapter contains examples using interactive operations see:

- *Examples of interactive operations* on page 8-10.

8.2 Using the Memory/Register Operations menu

Use the **Debug** menu to carry out read and write operations on memory and registers:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Debug** from the Code window main menu to display the **Debug** menu.
3. Select **Memory/Register Operations** to display the menu.

This menu contains:

Set Memory...

Displays the Interactive Memory Setting dialog box where you can walk through memory and make changes where required.

Patch Assembly...

Enables you to patch assembly code during your debugging session. You can enter instructions in assembler format for patching directly into memory. You can use labels, including making new ones, and symbols.

This is only available where supported by the underlying debug target. This option is disabled by default.

Set Register...

Displays the Interactive Register Setting dialog box where you can walk through registers and make changes where required.

Upload/Download Memory file...

Displays the Upload/Download file from/to Memory dialog box where you can locate a specific file, of a given type, and read the contents into an area of memory, or write a memory range into the file for re-use, or verify that a memory range matches the file contents.

Fill Memory with Pattern...

Displays the Fill Memory with Pattern dialog box where you can specify a pattern that is used to write to a given area of memory.

Flash Memory Control...

Displays the Flash Memory Control dialog box where you can erase and write Flash memory. The Flash memory must be opened before trying to use this dialog box.

8.3 Accessing interactive operations in other ways

There are other ways to access interactive operations depending on where you are working:

- *From the Memory pane*
- *From the Stack pane.*

8.3.1 From the Memory pane

If you are working in the Memory pane, you can access memory operations:

1. Select **View** → **Memory** to display the Memory pane.
2. Right-click on a memory cell, or byte, that is black or green, that is where the type is ROM, Flash, or modifiable, to display the **Memory** context menu.
3. Select the required option, for example **Set Memory Interactive...** to display the Interactive Memory Setting dialog box (see *Setting memory* on page 8-10 for instructions on how to use this dialog box).

8.3.2 From the Stack pane

If you are working in the Stack pane, you can access memory operations:

1. Select **View** → **Stack** to display the Stack pane.
2. Right-click on a memory cell, or byte, to display the **Memory** context menu.
3. Select the required option, for example **Set Memory Interactive...** to display the Interactive Memory Setting dialog box (see *Setting memory* on page 8-10 for instructions on how to use this dialog box).

8.4 Working with Flash

To use RealView Debugger to control Flash memory on your chosen debug target, you must:

- have access to an appropriate *Flash Method* (FME) file
- have access to a *Board/Chip Definition* (BCD) file that specifies the memory map of your debug target, and the FME file to use
- assign the BCD file to your debug target.

Depending on your current target, this might mean that you must first:

- create the FME file for your Flash type
- create and set up the BCD file.

For detailed instructions on creating an FME file, and creating, setting up and assigning a BCD file to your debug target, see Chapter 16 *Programming Flash with RealView Debugger*.

When you have configured your debug target to use the required BCD and FME files, you can program your Flash image into the Flash device. This section describes how to program a Flash image into the Flash device on your debug target. It includes:

- *Flash Method files*
- *Flash examples*
- *Flash programming* on page 8-6
- *Using the Flash Memory Control* on page 8-8.

8.4.1 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.

8.4.2 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\Core\...\flash\examples
```

Files are collected in subdirectories based on the target board or Flash device.

8.4.3 Flash programming

Before you can use RealView Debugger to control a Flash device on your target, you must:

- describe the Flash memory chip in the memory map
- ensure that you have a correctly configured FME file.

The following example describes how to program the Flash memory on an ARM® Integrator™/AP board that is connected using RVI/RVI-ME. It assumes that your debug target is correctly configured to use the BCD file and FME file for the Integrator/AP board. The Integrator/AP FME file is installed as part of the root installation, in the Flash examples directory ... \IntegratorAP (see *Flash examples* on page 8-5). Not all toolkits support Integrator/AP, but other BCD and FME files might be supplied for your supported target.

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 image 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.

Programming an image into Flash

To program an image, you ask RealView Debugger to write to the Flash memory region that you have defined in the board file. The Integrator/AP Flash starts at memory address 0x24000000.

The following procedure uses the Load File to Target dialog box to load a Flash image in preparation for writing to Flash. However, you can also program the image into Flash using the Upload/Download file to/from Memory dialog box. For instructions on how to do this, see *Loading Flash memory with the Upload/Download file from/to Memory dialog box* on page 8-18.

To write an image to the Integrator/AP Flash:

1. Build an image compiled to run with code at 0x24000000 and that has data in RAM.

This example uses the dhrystone program stored in:

`install_directory\RVDK\Examples\...\dhrystone`

To rebuild the dhrystone image with modified linker options:

- a. Open the dhrystone project.
 - b. Select **Project** → **Project Properties** to open the **Project Properties** window.
 - c. Click on the entry *BUILD in the left hand pane and double-click on the Link_Advanced entry.
 - d. Right-click on the Ro_base entry in the right-hand pane and select **Edit Value** from the context menu. Enter the value 0x24000000.
 - e. Right-click on the Rw_base entry in the right-hand pane and select **Edit Value** from the context menu. Enter the value 0x8000.
 - f. Select **Save and Close** in the Project Properties window.
Wait until RealView Debugger has finished generating the makefiles.
 - g. Select **Build** → **Rebuild All** to build the project.
2. Connect to the target device (Integrator/AP in this example).
 3. Click on the blue hyperlink in the File Editor pane to load the dhrystone.axf image.

This displays the Flash Memory Control dialog box (see Figure 8-1 on page 8-8). RealView Debugger queues the image in preparation for writing to Flash.

4. Click **Write** to program the image into Flash.

———— **Note** ————

Wait for the write to complete before continuing.

See *Using the Flash Memory Control* on page 8-8 for more details on how to use the Flash Memory Control dialog box.

5. Click **Close** to close the Flash Memory Control dialog box.
6. Click on the **Cmd** tab in the Output pane to see the Flash operations.
7. Select **View** → **Memory Map tab** to display the **Map** tab where you can see the Flash memory area (green icon) on the Integrator/AP board. Expand the Flash memory map entry to see the image and Flash details.

8.4.4 Using the Flash Memory Control

The Flash Memory Control dialog box shown in Figure 8-1 enables you to perform various operations on a Flash device.

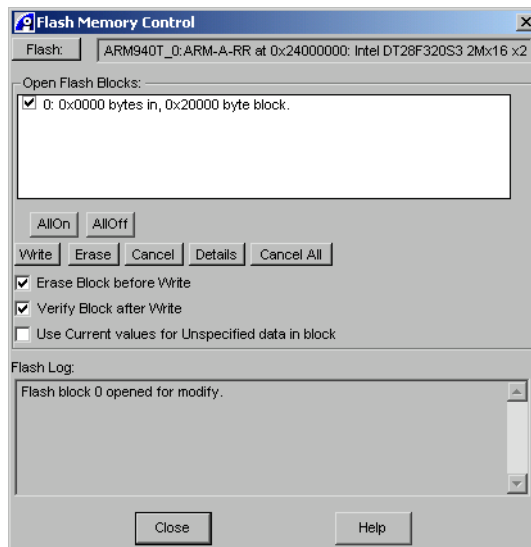


Figure 8-1 Flash Memory Control dialog box

The Flash Memory Control dialog box consists of a display list, a read-only data field, a Flash Log, and control buttons:

Flash: Click this button to view details about the Flash. The data field next to the **Flash** button describes the connection, Flash start address, and type of Flash being used.

Open Flash Blocks:

Lists the Flash blocks that are currently open for access. A check box is associated with each block. When this check box is checked, any Flash operations are performed on that block. When you first write an image to Flash, only those Flash blocks that are affected by the image are selected.

AllOn Selects all entries in the Open Flash Blocks list as indicated by a check in the accompanying check box. This enables you to carry out operations on all the open blocks.

AllOff Unselects all entries in the Open Flash Blocks list as indicated by no check in the accompanying check box.

- Write** Writes data to the specified blocks of Flash.
- Erase** Erases every specified block of Flash. This normally sets every byte to 0x00 or 0xFF depending on the type of Flash being used.
- Cancel** Abandons any changes made to the specified blocks of Flash.
- Cancel All** Abandons all changes to the Flash contents.
- Details** Displays an information box describing the type of Flash being used.
- Erase Block before Write**
Select this check box to erase the Flash block before performing the write operation.
- Verify Block after Write**
Select this check box to verify the Flash block, against the data source, after performing the write operation.
- Use Current values for Unspecified data in block**
Specifies that the original contents are to be maintained unless modified by the current operation. If unselected, the erase values are used.
Only use this option if you are updating part of the Flash block and you want to retain the current values in the rest of the block. This check box is unselected by default. If you select this option, RealView Debugger reads and then writes the entire block. This might take some time to complete.
- Flash Log:** Displays a log of operations carried out on the selected Flash blocks.
- Close** Closes the Flash Memory Control dialog box.
- Help** Displays online help for this dialog box.

See *Setting Flash memory* on page 8-20 for an example of interactive Flash memory operations.

8.5 Examples of interactive operations

This section contains examples showing how to perform interactive operations on memory and registers:

- *Setting memory*
- *Setting registers* on page 8-12
- *Downloading memory to a file* on page 8-14
- *Comparing memory with file contents* on page 8-16
- *Filling memory with a pattern* on page 8-16
- *Loading Flash memory with the Upload/Download file from/to Memory dialog box* on page 8-18
- *Setting Flash memory* on page 8-20.

8.5.1 Setting memory

To set memory contents:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Edit** → **Advanced** → **Show Line Numbers** to display line numbers.
This is not necessary but might help you to follow the examples.
3. Click on the **Src** tab to view the source file `dhry_1.c`.
4. Set a default breakpoint by double-clicking on line 301.
5. Click **Go** 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 red box marks the location of the PC when execution stops.
7. Select **View** → **Memory** to display the Memory pane.
Start addresses can be set using in-place editing or using the **Memory** context menu.
8. Right-click in the first address in the window to display the **Memory** context menu.
9. Select **Set Start Address...** and enter `0x000088A0` as the new start address.
10. Highlight the first byte at this address. In this example, this is `0x08`.

11. Right-click and select **Set Memory Interactive...** from the context menu to display the Interactive Memory Setting dialog box, shown in Figure 8-2 on page 8-11.

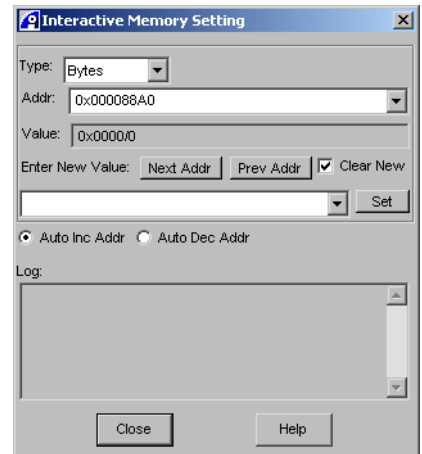


Figure 8-2 Interactive Memory Setting dialog box

12. Enter the required memory settings:
 - Type:** Select the display format. See *Formatting options* on page 7-13 for details of the memory formats.
 - Addr:** The address where the memory setting starts. Depending on the method used to display this dialog box, this field is already populated, as in this example.
The address must be entered in hexadecimal format, for example 0x000088A0.
 - Value:** This read-only data field shows the current value, in hexadecimal and decimal formats, at the specified memory location.
 - Enter New Value:**
Enter the value to be set at the current location, for example 0x08 or 8 (decimal).
If the Memory pane is configured to update automatically, clicking **Set** immediately updates the memory contents. This is the default setting in the **Pane** menu.
If you press Enter with no value in the Value data field, RealView Debugger moves automatically to the next, or previous, location.

Next Addr

Moves the target address to the next location by adding 1 to the address displayed in the Addr data field. This depends on the size of the current type.

Prev Addr

Moves the target address to a new location by subtracting 1 from the address displayed in the Addr data field. This depends on the size of the current type.

Clear New

Automatically clears any value entered in the Enter New Value data field ready to accept another value. By default, this feature is enabled.

Auto Inc Addr

If selected, this radio button instructs RealView Debugger to increment the target address automatically ready to accept a new setting.

Auto Dec Addr

If selected, this radio button instructs RealView Debugger to decrement the target address automatically ready to accept a new setting.

Log: Displays a log of the changes you have made. This log is shown when you next display the dialog box.

13. See the log updated to show your change.
14. Click **Close** to close the Interactive Memory Setting dialog box.
15. Double-click on the red marker disc to clear the breakpoint at line 301.

Changed values are displayed in the Memory pane in the usual way. That is, updated values are displayed in dark blue or light blue, depending on when they last changed.

8.5.2 Setting registers

To set register contents:

1. Select **Target** → **Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
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. Select **View** → **Registers** to display the Register pane.
7. Select **Debug** → **Memory/Register Operations** → **Set Register...** from the Code window main menu to display the Interactive Register Setting dialog box. This dialog contains almost the same controls as the Interactive Memory Setting dialog box described in *Setting memory* on page 8-10.
8. Set up the required register settings:
 - Register:** Enter the register to change, for example @R4. Press Enter to confirm your choice.
If required, use the drop-down arrow to select a previously used register from the stored list.
 - Value:** This read-only data field shows the current value, in hexadecimal and decimal formats, for the specified register.
 - Enter New Value:**
Enter the value to be set, in hex or decimal, for example 0xCC4.
All changed registers are displayed in blue.
 - Next Addr**
Moves to higher register by adding 1 to the register number displayed in the Register field.
 - Prev Addr**
Moves to a lower register by subtracting 1 from the register number displayed in the Register field.
 - Clear New**
Automatically clears any value entered in the Enter New Value data field ready to accept another value. By default, this feature is enabled.
 - Auto Inc Reg**
If selected, this radio button instructs RealView Debugger to increment the register number automatically ready to accept a new setting.
 - Auto Dec Addr**
If selected, this radio button instructs RealView Debugger to decrement the register number automatically ready to accept a new setting.
 - Log:** Displays a log of the changes you have made. This log is shown when you next display the dialog box.
9. See the log updated to show your change.

10. Click **Close** to close the Interactive Register Setting dialog box.
11. Double-click on the red marker disc to clear the breakpoint at line 301.

8.5.3 Downloading memory to a file

To download a memory range into a file:

1. Select **Target** → **Reload Image to Target** to reload the image `dhrystone.axf`.
2. Click on the **Src** tab to view the source file `dhry_1.c`.
3. Set a default breakpoint by double-clicking on line 301.
4. Click **Go** 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. 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, shown in Figure 8-3.

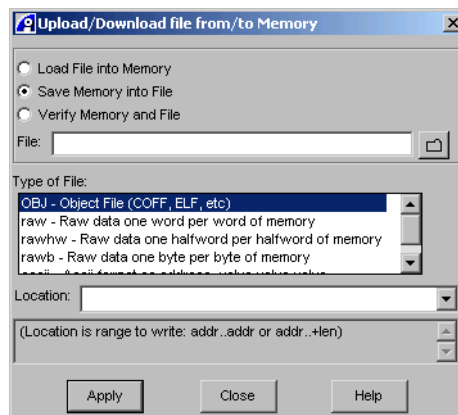


Figure 8-3 Upload/Download file from/to Memory dialog box

7. 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 pathname of the file to use to read/write memory values.

- c. In the **Type of File** section of the dialog, 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 a data file as a stream of 32-bit values
 - rawhw specifies a data file as a stream of 16-bit values
 - rawb specifies a data file as a stream of 8-bit values
 - ascii specifies a space-separated file of hexadecimal values.
- d. Define the start location of the memory block.

When writing memory, specify a range as an address range or as a start address and length, for example:

- 0x88A0..0x8980
- 0x88A0..+1000
- main..+1000

If required, use the drop-down arrow to select a previously used location from the stored list.

———— **Note** ————

If you are reading from a file to memory, you must specify a start location. The range can be left blank, unless the data type is binary.

If you are writing to a file from memory, you must specify a start location and a range.

8. Click **Apply** to create and write the specified file. If the specified file already exists, you are prompted to overwrite it.
9. Click **Close** to close the Upload/Download file to/from Memory dialog box.
10. Double-click on the red marker disc to clear the breakpoint at line 301.

———— **Note** ————

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.

8.5.4 Comparing memory with file contents

To verify a file:

1. Ensure that you have downloaded a memory range into a file as described in *Downloading memory to a file* on page 8-14.
2. 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 (see Figure 8-3 on page 8-14).
3. Select **Verify Memory and File**.
4. Specify the file to be compared.
5. Specify the start address to be compared. This 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 on the **Cmd** tab in the Output pane and view the results, for example:

```
verifyfile,ascii,gui "D:\ARM\RVD\Test_files\memory_file_3" Mismatch at  
Address 0x000088B6: 0x8E vs 0x8F
```

The first mismatch is identified and the location reported. Any mismatches after this location are not reported.
8. Click **Close** to close the Upload/Download file from/to Memory dialog box.
9. Double-click on the red marker disc to clear the breakpoint at line 301.

8.5.5 Filling memory with a pattern

To fill a specified area of memory with a predefined pattern:

1. Select **Target** → **Reload Image to Target** to reload the image *dhrystone.axf*.
2. Click on the **Src** tab to view the source file *dhry_1.c*.
3. Set a default breakpoint by double-clicking on line 301.
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. Select **View** → **Memory** to display the Memory pane.

7. Double-click in the address column of the Memory pane, and enter 0x88A0 to view the memory contents at that location.
8. Select **Debug → Memory/Register Operations → Fill Memory with Pattern...** from the Code window main menu to display the Fill Memory with Pattern dialog box, shown in Figure 8-4.

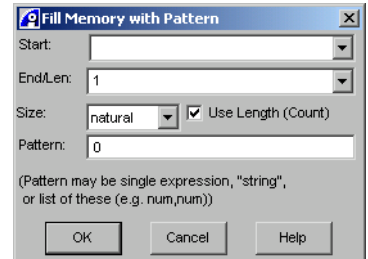


Figure 8-4 Fill Memory with Pattern dialog box

9. Set up the required memory settings:

Start: Enter the start address for the memory range to be filled, for example 0x88A0.

If required, use the drop-down arrow to select a previously used start address from the stored list.

End/Len: By default, the memory area that is filled is defined by a start address and a length. Enter the length of the memory block to be filled in this data field, for example 14 (decimal).

The specified length must be given relative to the data type given in the **Size** data field.

If the **Use Length (Count)** check box is unselected, you can specify the address that marks the end of the memory block.

Size: Enter the data type to be used in the file where:

 - `natural` indicates the format specified as native for the debug target
 - `byte` indicates support for 8-bit signed and unsigned byte form
 - `half-word` indicates support for 16-bit signed and unsigned halfwords
 - `long-word` indicates support for 32-bit signed and unsigned words.

Use Length (Count)

By default, the memory block to be filled is defined by a start address and the length. If this check box is unselected you can specify an address to mark the end of the filled block.

Pattern: Enter the pattern to be used as the fill, for example:

"AB"

0,1,0,1,0

———— **Note** ————

Strings must be enclosed in quotation marks.

10. Click **OK** to confirm your settings and close the Fill Memory with Pattern dialog box. Memory contents are rewritten and the Memory pane is automatically updated with changed values displayed in blue.
11. Double-click on the red marker disc to clear the breakpoint at line 301.

When filling memory blocks, you must be aware of the following:

- 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 and so might fill an area in excess of the specified block, for example specify a pattern of 10 bytes and a fill area of 16 bytes. RealView Debugger repeats the pattern twice and so fills a block of 20 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.

8.5.6 Loading Flash memory with the Upload/Download file from/to Memory dialog box

This example assumes that you have built the *dhrystone.axf* image as described in *Programming an image into Flash* on page 8-6, step 1, and that you are using the Integrator/AP board as the target.

To load Flash memory:

1. Connect to the Integrator/AP target.

2. 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, shown in Figure 8-3 on page 8-14.

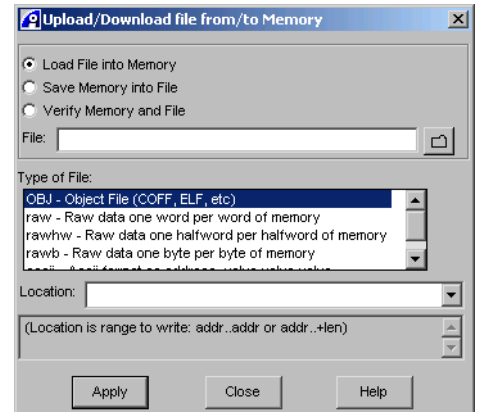


Figure 8-5 Upload/Download file from/to Memory dialog box

3. Specify the operation and set up the controls, as follows:
 - a. Select the **Load File into Memory** radio button. This instructs RealView Debugger to load the chosen file starting at the specified address.
 - b. In the **File** text box, enter the full pathname of the `dhrystone.axf` you built in step 1.
 - c. In the **Type of File** section of the dialog, select **OBJ**. This specifies an object file in the standard executable target format, for example ARM-ELF for ARM architecture-based targets.
 - d. Define the start location of the Flash memory.
For the Integrator/AP board, enter `0x24000000`.
4. Click **Apply**.
RealView Debugger queues the image in preparation for writing to Flash, and displays the Flash Memory Control dialog box (see *Using the Flash Memory Control* on page 8-8).

———— **Note** ————

Although RealView Debugger updates the Memory pane and the File Editor pane with the image information, the image is yet not written to the Flash.

5. Click **Write** to write the image to Flash.

Note

Wait for the write to complete before continuing.

6. Click **Close** to close the Flash Memory Control dialog box.
7. Click **Close** to close the Upload/Download file to/from Memory dialog box.

8.5.7 Setting Flash memory

Flash memory blocks open for access when you write to Flash, for example when you load an image. This displays the Flash Memory Control dialog box (see *Using the Flash Memory Control* on page 8-8 for details).

To write to Flash memory interactively:

1. Connect to your target, load an image, and write to Flash, as described in *Working with Flash* on page 8-5.
2. Select **View** → **Memory** to display the Memory pane.
Start addresses can be set using in-place editing or using the **Memory** context menu.
3. Right-click in the first address in the window to display the **Memory** context menu.
4. Select **Set Start Address...** and enter 0x24000000 as the new start address.
This is colored green, indicating Flash.
5. Right-click in the first byte at this address.
6. Select **Set Value...** from the context menu.
7. Enter the new value 0xA0 at the prompt.
8. Click **Set** to confirm this value. This displays the Flash Memory Control dialog box to enable you to access the open Flash, shown in Figure 8-1 on page 8-8.
9. If you want to view details of the Flash memory, click **Flash** in the Flash Memory Control dialog box. This displays details of the Flash memory, as shown in Figure 8-6 on page 8-21.

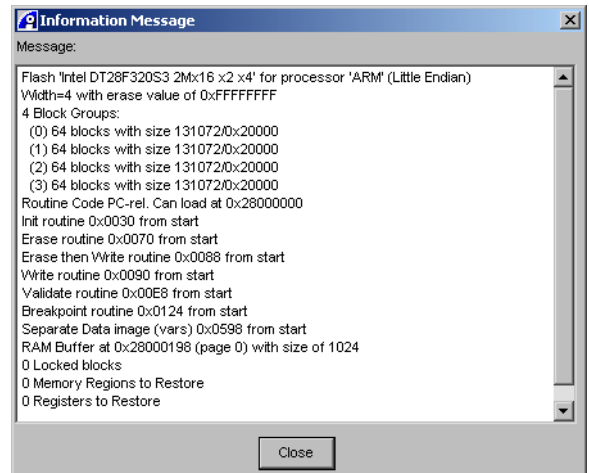


Figure 8-6 Flash memory details

You can also click **Details** to display information about the open Flash block.

Click **Close** to close the information box.

10. In the Flash Memory Control dialog box, ensure that the **Erase Block before Write** option is checked.
11. Click **Write** to write to the chosen Flash location. Monitor the changes in the Memory pane as memory is updated. The Flash Log confirms the Flash operation.
12. Click **Close** to close the Flash Memory Control dialog box.

Note

You can also select **Debug** → **Memory/Register Operations** → **Flash Memory Control...**, from the Code window main menu, to display the Flash Memory Control dialog box during your debugging session.

Chapter 9

Working with Browsers

RealView® Debugger provides browser panes and list dialog boxes to help with debugging tasks and monitor your program during execution. This chapter describes how to access these panes and dialog boxes from the Code window, and how to use them. It contains the following sections:

- *Using browsers* on page 9-2
- *Using the Data Navigator pane* on page 9-5
- *Using the Symbol Browser pane* on page 9-17
- *Using the Function List dialog box* on page 9-19
- *Using the Variable List dialog box* on page 9-21
- *Using the Module/File List dialog box* on page 9-23
- *Using the Register List Selection dialog box* on page 9-25
- *Specifying browser lists* on page 9-27
- *Using the Favorites Chooser/Editor dialog box* on page 9-29.

9.1 Using browsers

Browsers enable you to search through your source files to look for specific structures and to monitor their status during program execution.

This section includes:

- *Scope of symbols in RealView Debugger*
- *Browser panes*
- *List browser dialog boxes.*

9.1.1 Scope of symbols in RealView Debugger

RealView Debugger uses scope to determine the value of a symbol. Any symbol value available to a C or C++ program at the current PC is also available to RealView Debugger.

Variables can have values that are relevant within:

- a specific class only, that is *class scope*
- a specific function only, that is *local scope*
- a specific file only, that is *static global scope*
- the entire process, that is *global scope*.

For full details on scope and scoping rules see the chapter that describes working with the CLI in *RealView Developer Kit v2.2 Command Line Reference*.

9.1.2 Browser panes

The following browser panes are available:



- Data Navigator, to quickly locate a module, function, or variable (see *Using the Data Navigator pane* on page 9-5)
- Symbol Browser, for viewing C++ classes (see *Using the Symbol Browser pane* on page 9-17).

9.1.3 List browser dialog boxes

You can access list browser dialog boxes from various routes when working with RealView Debugger. These browsers enable you to select specific locations when you are performing operations such as:

- setting breakpoints
- setting watchpoints to monitor the contents of the specified location
- viewing the memory from the specified location.

To view a list browser:

1. Display the required pane, if the required pane is not in view:
 - **View** → **Break/Tracepoints** to display the Break/Tracepoints pane
 - **View** → **Watch** to display the Watch pane
 - **View** → **Memory** to display the Memory pane.
-  2. Click on the **Pane** menu button in the pane toolbar.
3. Select the required option:
 - On the Break/Tracepoints pane menu, select **Set/Edit Breakpoint...** to display the Set Address/Data Breakpoint dialog box. Also, see *Accessing the list browsers from the breakpoint dialog boxes* on page 9-4.
 - On the Watch pane menu, select **Enter New Expression...** to display the expression prompt box.
 - On the Memory pane menu, select **Set Start Address...** to display the expression prompt box.
-  4. Click on the drop-down arrow to the right of the field to display the options list menu.
5. Select the required browser from the menu:
 - <Function list...>**
Displays the Function List dialog box, described in *Using the Function List dialog box* on page 9-19.
 - <Variable list...>**
Displays the Variable List dialog box, described in *Using the Variable List dialog box* on page 9-21.
 - <Module list...>**
Displays the Module/File List dialog box, described in *Using the Module/File List dialog box* on page 9-23.
 - <Register list...>**
Displays the Register List dialog box, described in *Using the Register List Selection dialog box* on page 9-25. This option is only available from the various dialog boxes used to set breakpoints, see *Accessing the list browsers from the breakpoint dialog boxes* on page 9-4.

Because the browsers are used only to make a selection, there are no controls for debugging operations such as **Watch**, **Break**, or **Show Type Info**.

Accessing the list browsers from the breakpoint dialog boxes

The list browsers are also accessible from the following breakpoint dialog boxes:

- Simple Break if X (see *Using the Simple Break if X dialog box* on page 5-20)
- Simple Break if X, N times (see *Using the Simple Break if X, N times dialog box* on page 5-33)
- Simple Break if X, when Y is True (see *Using the Simple Break if X, when Y is True dialog box* on page 5-35)
- HW Break if in Range (see *Using the HW Break if in Range dialog box* on page 5-22)
- HW While in function/range, Break if X (see *Using the HW While in function/range, Break if X dialog box* on page 5-24)
- HW Break if X, then if Y (see *Using the HW Break if X, then if Y dialog box* on page 5-25)
- HW Break on Data Value match (see *Using the HW Break on Data Value match dialog box* on page 5-27).

9.2 Using the Data Navigator pane

The Data Navigator pane enables you to:

- quickly locate a module, function, or variable in one or more images that are loaded on the debug target
- perform various operations, such as setting breakpoints on functions or variables.

This section describes how to use the Data Navigator pane, and includes:

- *Displaying the Data Navigator pane*
- *Data Navigator pane interface components* on page 9-6
- *Applying a filter* on page 9-6
- *Refining the list of functions and variables* on page 9-8
- *Setting a default breakpoint on a function* on page 9-8
- *Setting a non-default breakpoint on a function* on page 9-9
- *Setting a hardware breakpoint on a variable* on page 9-9
- *Working with the Images tab* on page 9-9
- *Working with the Modules tab* on page 9-10
- *Working with the Functions tab* on page 9-11
- *Working with the Variables tab* on page 9-14.

9.2.1 Displaying the Data Navigator pane

To display the Data Navigator pane, shown in Figure 9-1, select **View → Data Navigator** from the Code window main menu.

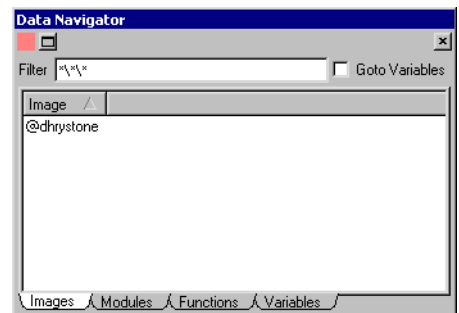


Figure 9-1 Data Navigator pane

9.2.2 Data Navigator pane interface components

The Data Navigator contains the following interface components:

Filter field Use this to quickly locate a function. This field is updated as you choose an image, module, and function. Alternatively, if you know the module and function name in the image, you can enter this directly.

See *Applying a filter* for more details on how to specify filters.

Goto Variables check box

By default, when you double-click on a module entry in the **Modules** tab, the **Functions** tab is selected. Check this box to have the **Variables** tab selected instead.

Images tab Lists the images that are loaded on the current debug target. See *Working with the Images tab* on page 9-9 for details on using this tab.

Modules tab Lists the modules in all images, or in the chosen image. See *Working with the Modules tab* on page 9-10 for details on using this tab.

Functions tab

Lists the functions in all images or modules, or in the chosen image or module. See *Working with the Functions tab* on page 9-11 for details on using this tab.

Variables tab

Lists the variables in all images or modules, or in the chosen image or module. See *Working with the Variables tab* on page 9-14 for details on using this tab.

9.2.3 Applying a filter

Using a filter enables you to narrow down the search when displaying the list of images, modules, functions, or variables in the Data Navigator pane. When you specify a filter, the list of functions and variables is filtered accordingly. That is, only those functions and variables that match the filter are displayed.

Filter syntax

The syntax for specifying a filter is:

- For modules:
`@image_name\module_name*`
 The `*` suffix is optional.

- For functions:
`@image_name\module_name\function_name`
- For variables:
`@image_name\module_name\variable_name`

Module names qualify symbolic references. See *Module naming conventions* on page 5-7 for details on using module names.

Filter metacharacters

Table 9-1 shows the metacharacters you can use to specify the filter rule or rules. When entering a filter, characters are case sensitive, for example the filter `*DHRY*` returns a list of three modules but `*dhry*` returns an empty list.

When you have completed the filter, press Enter and the list is refreshed. By repeatedly entering filters, and pressing Enter, you can refine the search to focus on selected modules, files, functions or variables.

Table 9-1 Filter metacharacters

Metacharacter	Description
<code>*</code>	This operator matches any character or number of characters, for example <code>*DHRY*</code> matches <code>MY_DHRYSTONE_H</code> but not <code>Dhrystone_H</code> or <code>MY_DHR</code> .
<code>?</code>	This operator matches any single character, for example <code>*DHRY_?</code> matches <code>MY_DHRY_A</code> but not <code>MY_DHRY_AB</code> or <code>DHRY_B</code> .
<code>[...]</code>	<i>List</i> operators enable you to define a set of items to use as a filter. The list items must be enclosed by square brackets, for example <code>*[HN]*</code> matches <code>DHRY_H</code> and <code>UNNAMED_1</code> but not <code>STDLIB</code> . An empty list (<code>[]</code>) returns no results.
<code>^</code>	This operator is used inside a list, to represent a NOT action, for example <code>*_[^2]*</code> matches <code>DHRY_1</code> but not <code>DHRY_2</code> .
<code>-</code>	<i>Range</i> operators enable you to define a range of items to use as a match. The range must be enclosed within square brackets, for example <code>*_[A-Z]*</code> matches <code>DHRY_H</code> but not <code>DHRY_1</code> whereas <code>*_[^A-Z]*</code> matches <code>UNNAMED_1</code> but not <code>DHRY_H</code> .
<code>-</code>	Used as the first character in a filter, this operator means do not match. For example, <code>*SAM* -*HOST*</code> means match all names containing the string <code>SAM</code> except those that contain the string <code>HOST</code> .

9.2.4 Refining the list of functions and variables

You can refine the list of functions and variables in the Data Navigator pane by using a filter (see *Applying a filter* on page 9-6). In addition, you can also use the following options on the **Functions** context menu (see *Functions context menu* on page 9-12) and the **Variables** context menu (see *Variables context menu* on page 9-15):

Show Publics

List global or public functions or variables with scope over all parts of the program.

Show Statics

List static functions or variables.

Shows Labels

List code labels with scope over entire functions.

Show Locals

Displays local variables with scope within a function. This option is available only on the **Variables** context menu.

Show Library Symbols

List library symbols depending on the state of the **Show Publics**, **Show Statics**, **Show Locals**, and **Show Labels** options.

9.2.5 Setting a default breakpoint on a function

In the Data Navigator pane, you can set a default breakpoint on a chosen function. To do this:

1. Locate the function in the **Functions** tab.
2. Right-click on the function entry.
3. Select **Break** to set a breakpoint.

This sets a default breakpoint on the function. RealView Debugger uses the BREAKINSTRUCTION command to set the breakpoint:

- If the function is in RAM, a software breakpoint is set.
- If the function is in Flash or ROM, a hardware breakpoint is set. A warning message might also be issued, depending on your debug target.

For more details on default breakpoints, see *Setting default breakpoints* on page 5-14.

9.2.6 Setting a non-default breakpoint on a function

If you want to use the Data Navigator pane set a non-default breakpoint on a function, that is you want to set a breakpoint that has qualifiers or actions, do the following:

1. Right-click on the function, and select one of the following options from the **Functions** context menu:
 - **Show in Disassembly**
 - **Show Source**
 - **Scope To.**
2. Set the required breakpoint as described in the following sections:
 - *Setting unconditional breakpoints explicitly* on page 5-20
 - *Setting hardware breakpoints explicitly* on page 5-22
 - *Setting conditional breakpoints* on page 5-32
 - *Using the Set Address/Data Breakpoint dialog box* on page 5-39.

9.2.7 Setting a hardware breakpoint on a variable

In the Data Navigator pane, you can set a hardware access, read, or write breakpoint on a chosen variable. To do this:

1. Locate the variable in the **Variables** tab.
2. Right-click on the variable entry to display the context menu.
3. Select **Break...** from the context menu.

This displays the Set Address/Data Breakpoint dialog box, with the Location field containing the reference to the selected variable. Only **HW Access**, **HW Read**, and **HW Write** breakpoint types are valid for variables. For more details on using this dialog box, see *Using the Set Address/Data Breakpoint dialog box* on page 5-39.

9.2.8 Working with the Images tab

The **Images** tab in the Data Navigator pane lists the images that you have loaded onto the current debug target.

Images tab operations

You can perform the following operations with the **Images** tab:

- Sort the image names in ascending or descending order. To do this, click on the **Images** column header.

- Narrow the search for a module, function, or variable to a specific image. To do this, double-click on the image name. The **Modules** tab becomes the current tab (see *Working with the Modules tab*). Also:
 - the **Functions** tab lists those functions in the image that you have elected to show in that tab
 - the **Variables** tab lists those variables in the image that you have elected to show in that tab.

9.2.9 Working with the Modules tab

The **Modules** tab in the Data Navigator pane lists the modules in all images by default, or in the image you selected from the **Images** tab (see *Working with the Images tab* on page 9-9). You can also limit the list of modules by specifying a filter (see *Applying a filter* on page 9-6).

Modules tab operations

You can perform the following operations with the **Modules** tab:

- Sort the modules in ascending or descending order of Module Name, Filename, or Image. To do this, click on the appropriate column header.
- Narrow the search for a function or variable to a specific module. To do this, double-click on the required module entry:
 - By default, the **Functions** tab becomes the current tab (see *Working with the Functions tab* on page 9-11). This tab lists those functions in the module that you have elected to show in that tab.
 - If you have checked the **Goto Variables** check box, the **Variables** tab becomes the current tab (see *Working with the Variables tab* on page 9-14). This tab lists those variables in the module that you have elected to show in that tab.
- Locate the the lowest address in memory that is occupied by the selected module, see *Modules context menu* on page 9-11 for details.

Modules context menu

To display the **Modules** context menu, right-click on a module entry. This menu has a single option:

Scope To Locates the lowest address in memory that is occupied by the chosen module in the current view of the File Editor pane. The address is that of the function with the lowest address in the module:

- If the **Dsm** tab is selected, the function is displayed in the disassembly view.
- If a source file tab is selected, for example dhry_1.c, and the chosen module matches that source file, the function is displayed.
If the chosen module matches a source file that is not selected, RealView Debugger selects the source file. If the source file is not already open, then RealView Debugger opens it.

If the selected module is in an ARM library, then RealView Debugger is unable to display the source, and displays the following message in the **Src** tab:

No source for context: *module\function*

module\function refers to the function in the module that has the lowest address in memory, for example, DIVISION_S__rt_sdiv, or <Unknown> if RealView Debugger is unable to determine the context.

In all cases where a module context can be determined, RealView Debugger also locates the module in the **Dsm** tab. Select the **Dsm** tab to view the start address for the module.

If RealView Debugger is unable to determine the context, then address 0x00000000 is chosen. For example, if you scope to a module that relates to a .h file, such as DHRY_H.

9.2.10 Working with the Functions tab

The **Functions** tab in the Data Navigator pane lists the functions in all modules by default, or in the module you selected from the **Modules** tab (see *Working with the Modules tab* on page 9-10). You can also limit the list of functions displayed by specifying a filter (see *Applying a filter* on page 9-6).

Functions tab operations

You can perform the following operations with the **Functions** tab:

- Sort the functions in ascending or descending order of Function Name, Address, Scope, Module, or Image. To do this, click on the appropriate column header.

- Various operations are available from the **Functions** context menu. To display this context menu, right-click on a function entry. See *Functions context menu* for a description of these operations.

Functions context menu

The **Functions** context menu contains the following options:

Show Publics

When enabled (checked), lists all global or public functions with scope over all parts of the image, or selected module.

Show Statics When enabled (checked), lists all static functions in the image, or selected module.

Show Labels When enabled (checked), lists all code labels with scope over the entire image, or selected module.

Show Library Symbols

When enabled (checked), lists the symbols in the libraries used by the image. The library symbols listed depends on the state of the **Show Publics**, **Show Statics**, and **Show Labels** options.

When disabled (unchecked), most library symbols are filtered out. This is the default.

Show in Disassembly

Locates the function in the disassembly view in the **Dsm** tab of the File Editor pane. If the disassembly view is not visible, RealView Debugger switches the code view to the **Dsm** tab.

You can also locate the function in the disassembly view by double-clicking on the function entry.

Show Source

Locates the function in your source code of the File Editor pane. If the source file is not open, RealView Debugger opens the file. If the source file is open, but not visible, RealView Debugger switches the code view to the source.

Break

Sets a default breakpoint on the function using the `BREAKINSTRUCTION` command (see *Setting a default breakpoint on a function* on page 9-8).

If you want to set a non-default breakpoint on a function, that is you want to set a breakpoint that has qualifiers or actions, see *Setting a non-default breakpoint on a function* on page 9-9.

Go until Sets a temporary breakpoint at the specified function. The program then executes from the current position of the PC. When execution reaches the breakpoint it stops.

The temporary breakpoint exists only for the duration of this run and so is not shown in the Break/Tracepoints pane. Similarly, there is no red breakpoint marker shown in the source file. If the program stops before it reaches the temporary breakpoint, you must reinstate it before restarting the run.

This is equivalent to the CLI command:

```
go function
```

For example, go @dhystone\\DHR_1\Proc_2.

Show Type Info

Prints the type information for the selected function in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

```
printtype function
```

For example, printtype @dhystone\\DHR_1\Proc_2.

Show Full Info

Submits a CEXPRESSION command to calculate the value of a given expression by calling the specified target function.

The function is converted into a debugger call macro, and strings and arrays passed to the target function are copied onto a stack maintained by RealView Debugger. A function called in this way behaves as though it had been called from your program.

————— **Note** —————

Target calls are not supported by all debug environments.

Results are displayed in either floating-point format, address format, or in decimal, hexadecimal, or ASCII format depending on the type of variables used in the expression.

Set PC to Function

Submits a SETREG command to set the PC register to the start address of the selected function. For example:

```
setr @PC=@dhystone\\DHR_1\Proc_1
```

The Code window scrolls to the specified function and the red box shows the location of the PC.

- Scope To** Locates the start of the function in the current view of the File Editor pane:
- If the **Dsm** tab is selected, the function is displayed in the disassembly view.
 - If a source file tab is selected, for example `dhry_1.c`, and the chosen function is in that source file, that function is displayed.
If the chosen function matches a source file that is not selected, RealView Debugger selects the source file. If the source file is not already open, then RealView Debugger opens it.
- If the selected function is in an ARM library, then RealView Debugger is unable to display the source, and displays the following message in the **Src** tab:
- No source for context: *module\function*
module\function is the function reference, for example, `STDIO\fprintf`.
- In all cases, RealView Debugger also locates the function in the **Dsm** tab. Select the **Dsm** tab to view the start address for the function.

9.2.11 Working with the Variables tab

The **Variables** tab in the Data Navigator pane lists the variables in all functions by default, or in the module you selected from the **Modules** tab (see *Working with the Modules tab* on page 9-10). You can also limit the list of variables displayed by specifying a filter (see *Applying a filter* on page 9-6).

Variables tab operations

You can perform the following operations with the **Variables** tab:

- Sort the variables in ascending or descending order of Variable Name, Address, Scope, Module, or Image. To do this, click on the appropriate column header.
- Print the hexadecimal value of the variable to the **Cmd** tab of the Output pane. To do this, double-click on the variable entry.
- Various operations available from the **Variables** context menu. To display this context menu, right-click on a variable entry. See *Variables context menu* on page 9-15 for a description of these operations.

Variables context menu

The **Variables** context menu contains the following options:

Show Publics

When enabled (checked), lists all global or public variables with scope over all parts of the image, or selected module.

Show Statics When enabled (checked), lists all static variables in the image, or selected module.

Show Labels When enabled (checked), lists all code labels with scope over the entire image, or selected module.

Show Locals When enabled (checked), lists all local variables in the image, or selected module.

Show Library Symbols

When enabled (checked), lists the symbols in the libraries used by the image. The library symbols listed depends on the state of the **Show Publics**, **Show Statics**, **Show Labels**, and **Show Locals** options.

When disabled (unchecked), most library symbols are filtered out. This is the default.

Print Hex Prints the hexadecimal value of the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

```
print /h variable
```

For example, print `/h @dhystone\\Ch_1_Glob`.

You can also display the hexadecimal value by double-clicking on the variable entry.

Print Decimal

Prints the decimal value of the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

```
print variable
```

For example, print `@dhystone\\Ch_1_Glob`.

Watch Adds a watch for the selected variable.

Show Type Info

Prints the type information for the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

```
printtype variable
```

For example, `printtype @dhystone\\Ch_1_Glob`.

Show Full Info

Prints the full information for the selected variable in the **Cmd** tab of the Output pane. This is equivalent to the CLI command:

`printsym variable`

For example, `printsym @dhystone\\Ch_1_Glob`.

Break...

Displays the Set Address/Data Breakpoint dialog box, where you can set a hardware breakpoint (see *Setting a hardware breakpoint on a variable* on page 9-9).

9.3 Using the Symbol Browser pane

Use the Symbol Browser pane to examine C++ classes in your application program.

To examine C++ classes either:

- Select the menu option **View → Symbol Browser** from the Code window main menu.
- Move the focus to the chosen pane, for example the Watch pane, click the **Pane Control** menu, and select **New Pane → Symbol Browser** from the available options.

An example display is shown in Figure 9-2.

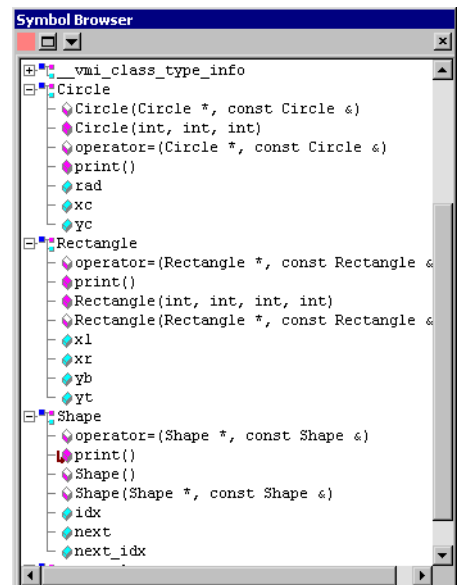


Figure 9-2 C++ symbols in the Symbol Browser pane

This section describes:

- *Viewing details of a class* on page 9-18
- *Viewing details of a function* on page 9-18.

9.3.1 Viewing details of a class

Colored icons are used to identify different components within a class:



Filled stack + arrow

This magenta icon indicates a function which is a declared member of the parent class.



Filled stack The magenta filled stack indicates that a member function of the parent class is both declared and defined. These members are real in that they are called during execution.



Hollow stack

Where magenta stacks are hollow, they indicate that a member function of the parent class is declared but not defined. These members are virtual in that they are not called during execution.



Filled block The filled blue block indicates a data object that is manipulated by a class function using operators, or methods, defined in the class.

Right-click on a chosen class to display the **Class** menu. This contains:

Find Class Definition...

Displays the Find in Files dialog box where you can specify the class details and so locate the definition in your source code.

Use this dialog box to locate class definitions when these are not provided by the compiler.

See the chapter that describes searching in *RealView Developer Kit v2.2 Project Management User Guide* for details on using the Find in Files dialog box.

Properties... Displays a text description of the item under the cursor.

9.3.2 Viewing details of a function

Right-click on a function to display the **Function** menu. This contains:

Show Function Definition

Scopes to the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Set Break Sets a breakpoint at the selected function. This option is enabled for defined functions identified by a magenta filled stack icon.

Properties... Displays a text description of the item under the cursor.

9.4 Using the Function List dialog box

Use the Function browser to examine the different functions that make up your program.

This section describes:

- *Displaying the Function List dialog box*
- *Specifying the list*
- *Refining the list* on page 9-20
- *Selecting a function* on page 9-20
- *Closing the browser* on page 9-20.

9.4.1 Displaying the Function List dialog box

Display the Function List dialog box, shown in Figure 9-3, as described in *List browser dialog boxes* on page 9-2.

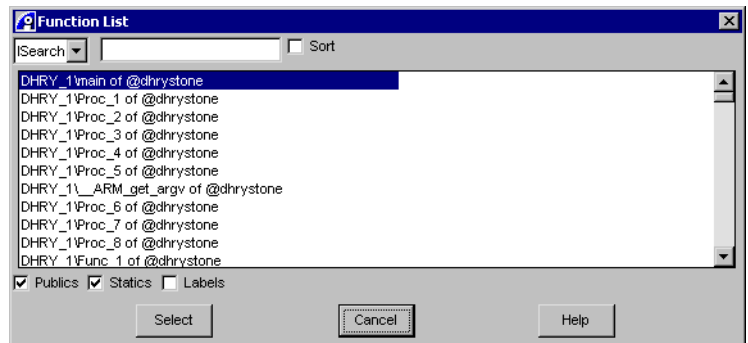


Figure 9-3 Function List dialog box

The Function List dialog box lists all the functions, ordered by module name, in the current program. Each entry in the list shows the filename, if known, and then the function name, for example:

DHR_Y_2\Func_3 of @dhrystone

9.4.2 Specifying the list

When you first open the Function List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which functions are displayed by applying a search filter.

See *Specifying browser lists* on page 9-27 for details of how to specify the list for the chosen browser.

9.4.3 Refining the list

The Function List dialog box contains check boxes that enable you to refine what is displayed in the list box:

Publics	Displays global or public functions with scope over all parts of the program.
Statics	Displays static functions.
Labels	Displays code labels with scope over the entire function.

9.4.4 Selecting a function

With a list of functions displayed in the Function List, select a function entry and click the **Select** button to select that function.

9.4.5 Closing the browser

When you click **Select**, the Function List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser.

9.5 Using the Variable List dialog box

Use the Variables browser to examine the different variables used in your program.

This section describes:

- *Displaying the Variable List dialog box*
- *Specifying the list* on page 9-22
- *Refining the list* on page 9-22
- *Closing the browser* on page 9-22.

9.5.1 Displaying the Variable List dialog box

Display the Variable List dialog box, shown in Figure 9-4, as described in *List browser dialog boxes* on page 9-2.

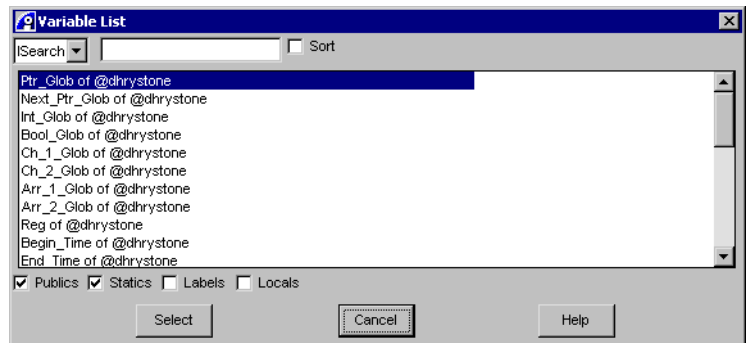


Figure 9-4 The Variable List

The Variable List dialog box shows all the variables in the current program. By default, the **Sort** check box is unselected and so the variables are given in order of occurrence.

Each entry in the list shows the variable name followed by the program name attached using @, for example:

Ch_1_Glob of @dhrystone

Including local variables adds the filename, if known, to the list, for example:

DHRY_2\Proc_8\Int_Index local of @dhrystone

9.5.2 Specifying the list

When you first open the Variable List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which variables are displayed by applying a search filter.

See *Specifying browser lists* on page 9-27 for details of how to specify the list for the chosen browser.

9.5.3 Refining the list

The Variable List dialog box contains check boxes that enable you to refine what is displayed in the list box:

Publics	Displays global or public variables with scope over all parts of the program.
Statics	Displays static variables.
Labels	Displays code labels with scope over the entire function.
Locals	Displays local variables with scope within the current function.

9.5.4 Closing the browser

When you click **Select**, the Variables List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser.

9.6 Using the Module/File List dialog box

Using the Module/File browser enables you to examine the different files and modules that make up your program and how these components are accessed during program execution. In this way you can locate errors during your debugging session.

This section describes:

- *Displaying the Module/File List dialog box*
- *Specifying the list on page 9-24*
- *Selecting a module on page 9-24*
- *Closing the browser on page 9-24.*

9.6.1 Displaying the Module/File List dialog box

Display the Module/File List dialog box, shown in Figure 9-5, as described in *List browser dialog boxes* on page 9-2.

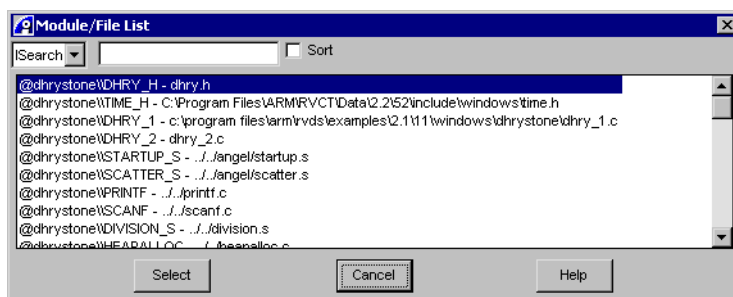


Figure 9-5 Module/File List dialog box

The Module/File List dialog box displays, in order of appearance, all the modules and files in the current program. Each entry in the list shows the module name and then the filename, if known, for example:

```
@dhrystone\DHRY_2 - dhry_2.c
```

The program name is attached at the start using @, for example @dhrystone\.

Module names qualify symbolic references. The module name is usually the filename without the extension. All module names are converted to uppercase by RealView Debugger. If the extension is not standard, the extension is preserved, and the dot is replaced with an underscore, for example `sample_arm.c` is converted to `SAMPLE_ARM`, and `sample_arm.h` is converted to `SAMPLE_ARM_H`.

If two modules have the same name then RealView Debugger appends an underscore followed by a number to the second module, for example `SAMPLE_1`. Additional modules are numbered sequentially, for example, if there is a third module this becomes `SAMPLE_2`.

Following this convention avoids any confusion with the `C` dot operator indicating a structure reference.

9.6.2 Specifying the list

When you first open the Module/File List dialog box, the list entries are determined by the default search entry **ISearch** but you can decide which modules and files are displayed by applying a search filter.

See *Specifying browser lists* on page 9-27 for details of how to specify the list for the chosen browser.

9.6.3 Selecting a module

To select a module:

1. Display the Module/File List dialog box as described in *List browser dialog boxes* on page 9-2.
2. Select a module from the list.
3. Click **Select** to to that module.

9.6.4 Closing the browser

When you click **Select** the Module/File List dialog box closes automatically. Otherwise, click **Cancel** to exit the browser without adjusting the scope.

9.7 Using the Register List Selection dialog box

The Register List Selection dialog box enables you to select a register from a list of memory mapped registers for the current target connection. The list of registers is generated from those defined in the Peripherals group of the Board/Chip Definition file assigned to the connection.

Note

You must assign a Board/Chip Definition file (for example, AP.bcd) to the connection to use this dialog box.

This section describes:

- *Displaying the Register List Selection dialog box*
- *Selecting a register on page 9-26*
- *Closing the browser on page 9-26.*

9.7.1 Displaying the Register List Selection dialog box

Display the Register List Selection dialog box, shown in Figure 9-6, as described in *List browser dialog boxes* on page 9-2.

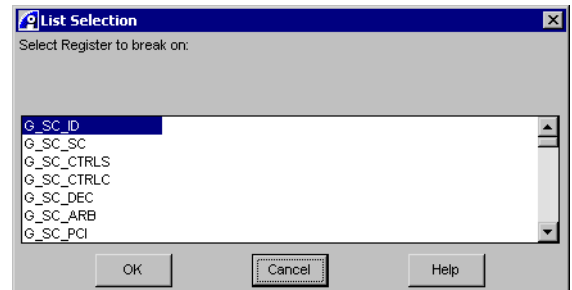


Figure 9-6 Register List Selection dialog box

The Register List Selection dialog box displays, in alphabetical order, the memory mapped registers. These registers are defined any Board/Chip Definitions you have assigned to the current target connection. The registers shown in Figure 9-6 are those for the Integrator/AP board, which are defined in the AP.bcd.

9.7.2 Selecting a register

To select a register:

1. Display the Register List Selection dialog box as described in *List browser dialog boxes* on page 9-2.
2. Select a register from the list.
3. Click **OK** to select that register.

9.7.3 Closing the browser

When you click **OK** the Register List Selection dialog box closes automatically. Otherwise, click **Cancel** to exit the browser without adjusting the scope.

9.8 Specifying browser lists

When you first open a browser, the list entries shown in the dialog box are determined by the default search entry **ISearch** but you can decide what contents are displayed by applying a search filter. To change the search mechanism either:

- click on the drop-down arrow to display the search list and select a search
- highlight the contents, for example **ISearch**, and press F or I to toggle the contents.

This section describes:

- *Specifying a list*
- *Applying a filter* on page 9-28.

9.8.1 Specifying a list

When you change the browser search mechanism to specify a new list, RealView Debugger performs the match against list entry *segments* and not against the text string as shown in the display list. This applies if you choose to sort the display list (see the description of **Sort** in this section) or to apply a filter to limit the search (see *Applying a filter* on page 9-28 for details).

To specify the search:

List of entries displayed

The list of entries displayed can be specified using one of the following:

- | | |
|----------------|---|
| ISearch | With this selected, the list of entries is created using the default search mechanism based on the names of the modules, files, functions, or variables found in your program. Enter a partial name in the Text entry field to move the highlight to the first matching occurrence. The search is case insensitive. |
| Filter | Use a filter to limit the search to list only those symbols that match certain criteria, see <i>Applying a filter</i> on page 9-28 for details. |

Text entry field

Enter here the filter to be used in the search and then press Enter. The filter enables you to search using the metacharacters listed in Table 9-1 on page 9-7. You can enter a list of search rules by combining different operators, see *Applying a filter* on page 9-28 for details of how to apply a search filter.

Sort Select this check box to order the display list in alphabetical order based on the names of the modules, files, functions, or variables found in your program. The sort is case insensitive, that is uppercase and lowercase are treated as identical. Sort is unchecked by default.

Note

If you change the default search entry to **Filter** and then enter a filter to set the display criteria, these settings are maintained when you close, or cancel, the browser.

9.8.2 Applying a filter

Using a filter enables you to narrow down the search when displaying the list of modules, files, functions, or variables. Table 9-1 on page 9-7 shows the metacharacters you can use to specify the filter rule or rules. When entering a filter, characters are case sensitive, for example the filter *DHR* returns a list of three modules but *dhr* returns an empty list.

When you have completed the filter, press Enter and the list is refreshed. By repeatedly entering filters, and pressing Enter, you can refine the search to focus on selected modules, files, functions or variables.

9.9 Using the Favorites Chooser/Editor dialog box

This section describes how to use the Favorites Chooser/Editor dialog box. It includes:

- *Overview of the Favorites Chooser/Editor dialog box*
- *Displaying the Favorites Chooser/Editor dialog box* on page 9-30
- *Favorites Chooser/Editor dialog box interface components* on page 9-30
- *Favorites categories used by RealView Debugger features* on page 9-32.

9.9.1 Overview of the Favorites Chooser/Editor dialog box

The Favorites Chooser/Editor dialog box enables you to save expressions that you use on a regular basis to a personal Favorites List. The following types of expression can be set as favorites:

- addresses and address ranges
- data values
- breakpoints
- qualifiers and actions for breakpoints
- watch expressions
- directories
- filenames.

When you first start to use RealView Debugger on Windows, your personal Favorites List is empty. You can create expressions and add them directly to this list or you can add expressions that you have been using in the current debugging session.

If you create an expression manually, you must make sure that the expression matches the Favorites Category (see *Favorites Chooser/Editor dialog box interface components* on page 9-30). For example, you must add an address range to the Addresses category, but not to the Break/Tracepoints category. If you do add a favorite to the wrong category, then RealView Debugger might not behave as expected, or an error message might be displayed.

RealView Debugger keeps a record of all favorite expressions that you set during your debugging session as part of your history file `exphist.sav`, which is located in your RealView Debugger home directory. This file also contains a history of other entities, for example files and directories you have accessed during the debugging session. By default, at the end of your debugging session, these processor-specific lists are saved in the history file.

————— Note —————

You must connect to a target to enable RealView Debugger favorites.

9.9.2 Displaying the Favorites Chooser/Editor dialog box

There are many ways you can display the Favorites Chooser/Editor dialog box in RealView Debugger:

- From any dialog boxes that have a drop-down arrow to the right of the various fields, for example, the various breakpoint dialog boxes.
- By selecting the appropriate favorites option from a menu, for example, the **Set from Favorites** option on the context menu of the Register pane.

The Favorites Chooser/Editor dialog box is shown in Figure 9-7.

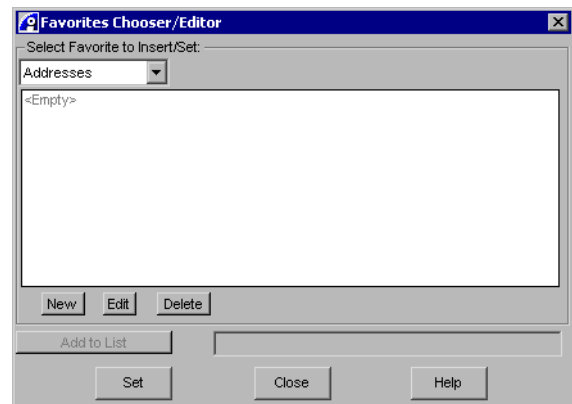


Figure 9-7 Favorites Chooser/Editor dialog box

9.9.3 Favorites Chooser/Editor dialog box interface components

The main interface components of the Favorites Chooser/Editor dialog box (see Figure 9-7) are:

Favorites Category

This is the category that is to contain the favorite expression. *Favorites categories used by RealView Debugger features* on page 9-32 identifies the dialog boxes where each categories are used.

RealView Debugger automatically sets this to the context from which you displayed the dialog box. For example, if you want to set a breakpoint from the Break/Tracepoint favorites, RealView Debugger sets this to Break/Tracepoints.

Alternatively, you can select another category if you want to add an expression manually. Use the **New** button to create a new favorite yourself.

Favorites List

	Lists any favorites that are currently defined for the selected Favorites Category.
New	Click this button to manually create a new favorite. When you click this button a New/Edit Favorite dialog box is displayed. Enter the required expression and a meaningful description. The expression must match the Favorites Category, for example, a breakpoint if the Break/Tracepoint category is selected. When you have finished, click OK to add the expression to the Favorites List.
Edit	Click this button to edit the selected favorite.
Delete	Deletes the selected favorite from the Favorites List.
Add to List	<p>This button is enabled if an expression is shown in the text box to the right of the button. Otherwise it is disabled.</p> <p>When you click this button a New/Edit Favorite dialog box is displayed showing the expression from the text box. You can also enter a meaningful description for the expression. When you have finished, click OK to add the expression to the Favorites List.</p>
Set	Closes the dialog, and sets the selected favorite for the current context.
Close	Closes the dialog box without using a favorite.
Help	Displays online help for this dialog box.

9.9.4 Favorites categories used by RealView Debugger features

Table 9-2 describes where each Favorites Category is used in RealView Debugger.

Table 9-2 Favorites Categories used by RealView Debugger features

Favorites Category	Where it is used in RealView Debugger
Addresses	<p>This is used where you can specify an address or an address range:</p> <ul style="list-style-type: none"> • In the dialog boxes for setting breakpoints, where you want to specify an address. See Chapter 5 <i>Working with Breakpoints</i> for details on using the breakpoint dialog boxes. • In the dialog boxes used to locate a specific address, such as in the: <ul style="list-style-type: none"> — Stack pane (see <i>Using the Stack pane</i> on page 7-23) — Memory pane (see <i>Displaying memory contents</i> on page 7-11) — Dsm tab.
Data Values	<p>Used to specify data values in the dialog boxes when setting:</p> <ul style="list-style-type: none"> • Breakpoints See Chapter 5 <i>Working with Breakpoints</i> for details on using the breakpoint dialog boxes. • The value of a memory location in the Memory pane. See <i>Displaying memory contents</i> on page 7-11. • The value of a register in the Register pane. See <i>Changing register contents</i> on page 7-6.
Data Ends	<p>Used by the HW Break on Data Value match dialog box, to specify a data value modifier. See <i>Using the HW Break on Data Value match dialog box</i> on page 5-27 for details.</p>
Watch Expressions	<p>Used for specifying watchpoints in the Watch pane. See <i>Working with watches</i> on page 7-32 for details on setting watchpoints.</p>
Break/Tracepoints	<p>This is used to specify CLI commands for setting breakpoints:</p> <ul style="list-style-type: none"> • For details on setting breakpoints, see Chapter 5 <i>Working with Breakpoints</i>.
Break Qualifiers	<p>This is used for breakpoint qualifiers on the Set Address/Data Breakpoint dialog box. See <i>Specifying Qualifiers</i> on page 5-48 for details.</p>

Table 9-2 Favorites Categories used by RealView Debugger features

Favorites Category	Where it is used in RealView Debugger
Break Actions	This is used for breakpoint actions on the Set Address/Data Breakpoint dialog box. See <i>Specifying Actions</i> on page 5-49 for details.
Filenames	This is used for the various open and save file dialog boxes to specify files, such as opening source files, include and workspace files, and selecting log files.
Directories	This is used for the various open and save file dialog boxes to specify directories, such as opening source files, include and workspace files, and selecting log files.

Chapter 10

Working with Macros

In RealView® Debugger, a macro is a C-like function that is invoked by entering a single command using the macro name. This section describes how to define macros for use during your debugging session, and how to save and edit your macros. It contains the following sections:

- *About macros* on page 10-2
- *Using macros* on page 10-8
- *Getting more information* on page 10-17.

10.1 About macros

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), and 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 a breakpoint to create breakpoint condition testing
- an attachment to a window or file where the macro can send information.

———— **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.

This section gives an overview of macros in RealView Debugger. It includes the following sections:

- *Disconnect (Defining Mode)...* on page 2-22
- *Debugger commands in macros* on page 10-3
- *Defining macros* on page 10-4
- *Calling macros* on page 10-5
- *Macro return values* on page 10-5
- *Using macros with breakpoints* on page 10-6
- *Attaching macros* on page 10-7
- *Stopping macros* on page 10-7.

10.1.1 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 commands
- be defined in a debugger include file.

Macros cannot:

- be recursive
- define global variables
- define static variables
- define other macros.

10.1.2 Debugger commands in macros

RealView Debugger enables you to enter debugger commands on the command line, or when using the headless debugger, for example PRINTF or SETMEM.

You can define a macro that contains a sequence of debugger commands. When used in this way, the command must be enclosed by dollar signs (\$), shown in Example 10-1.

Example 10-1 Using debugger commands in macros

```
some_commands
define int registers()
{some_commands
for(macro_index = 0; macro_index < 6; macro_index++) {
macro_bin_str[macro_index + 8] = macro_cpsr_mode[macro_index +
(6*macro_cpsr_key)]; } macro_bin_str[14] = 0x00; $printf " r0 = 0x%x" ,@r0$;
$printf " r1 = 0x%x" ,@r1$; $printf " r2 = 0x%x" ,@r2$;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.

The following commands cannot be used inside a macro:

- ADD
- DEFINE (unless it is the macro definition itself)
- DELETE
- HELP
- HOST
- INCLUDE
- QUIT.

Because macros can return a value, they can also be used in expressions. When the macro executes, the return value is used according to the return type.

For full details on all CLI commands, see *RealView Developer Kit v2.2 Command Line Reference*.

Attaching macros

Macros can also be invoked as actions associated with:

- a window
- a breakpoint
- deferred commands, for example BGLOBAL.

In this case, execution-type commands cannot be used inside a macro:

- GO
- GOSTEP
- STEPLINE, STEPO
- STEPINSTR, STEPPOINSTR.

If you require a conditional breakpoint that performs an action and then continues program execution, you must use the breakpoint continue qualifier, or return 1 from the macro call, instead of the GO command.

10.1.3 Defining macros

You can define a macro outside RealView Debugger using a text editor and load the macro definition file in with the INCLUDE command. The number of macros that can be defined is limited only by the available memory on your workstation.

You can also create, edit, save, and delete macros from the Code window using the **Debug** menu, as demonstrated in *Using macros* on page 10-8.

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.

10.1.4 Calling macros

Macros are called from the RealView Debugger command line. The call consists of the macro name followed by a set of parentheses containing the macro arguments, separated by commas.

Macro names are case sensitive and must be entered as shown in the definition. The macro arguments are converted to the types specified in the macro definition. If RealView Debugger cannot convert the arguments it generates an error message.

Examples of macro calls are:

mytext(var) Calls the macro named `mytext` with the argument `var`.

count(7) Calls the macro named `count` with the parameter `7`.

You can define a macro with a name that is identical to a command or keyword used by RealView Debugger. You can then use the macro name in an expression and submit it on the command line where it is interpreted correctly.

If, however, you submit the macro name as a command, RealView Debugger cannot identify it as a macro. To overcome this, use the prefix `MACRO` when entering macro names that might conflict with debugger keywords or command names:

```
MACRO macro_name()
```

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 qualified. For example, `strcpy` is a predefined debugger macro, while `PROG\strcpy` is a function within the module `PROG`. The predefined macro is referenced as `strcpy(dest,src)`, while `PROG\strcpy(dest,src)` refers to the function within `PROG`.

10.1.5 Macro return values

The type of the macro return value is specified by `return_type` when you define the macro. If `return_type` is not specified, then type `int` is assumed.

When attached to a breakpoint, the macro return value controls the behavior of the breakpoint, see *Using macros with breakpoints*.

10.1.6 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 qualifier (see *Setting conditional breakpoints* on page 5-32). Therefore, you can test your program variables and decide how the breakpoint behaves when it is hit (see *Controlling breakpoint behavior with macro return values* on page 10-7).

The frequency that a macro runs when the breakpoint is hit depends on the order it appears with other condition qualifiers. See *Controlling the behavior of breakpoints* on page 5-55 for more details.

You can use conditional statements in your macro to change the execution path when the breakpoint is triggered 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.

For an example of attaching a macro to a breakpoint and using it to control program execution see *Attaching macros to breakpoints* on page 5-52.

————— **Note** —————

You cannot use the `GO`, `GOSTEP`, `STEPINSTR`, `STEPLINE`, `STEP0`, or `STEP0INSTR` commands in a macro that is attached to a breakpoint. If any of these commands are present in a such a macro, the following messages are displayed:

```
Error: Cannot perform operation - thread is running.
Error: E0081: Runtime error in macro.
```

Controlling breakpoint behavior with macro return values

You can use macro return values to control what action RealView Debugger takes when a breakpoint is triggered:

- If the macro returns a nonzero value (False), RealView Debugger continues program execution. Any qualifiers and actions that appear after the macro are not processed, and the triggering of the breakpoint is not recorded.
- If a macro returns a value of zero (True), 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 *Setting conditional breakpoints* on page 5-32 for more details.

10.1.7 Attaching macros

In addition to breakpoints, you can attach macros to:

- aliases, for example `ALIAS my_alias=my_macro()`
- windows, for example `VMACRO 250, my_macro()`.

10.1.8 Stopping macros



When macros are run as commands they are queued for execution like any other debugger command when your program is executing. Click the **Command cancel** toolbar button 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 the **Stop Execution** button to stop a macro that is attached to a breakpoint.

10.2 Using macros

This section shows you how to start using macros by working through an example. It contains the following sections:

- *Creating a macro*
- *Viewing a macro* on page 10-10
- *Testing a macro* on page 10-10
- *Editing a macro* on page 10-11
- *Copying a macro* on page 10-13
- *Deleting a macro* on page 10-14
- *Calling a macro* on page 10-14.

10.2.1 Creating a macro

To create a macro from within RealView Debugger, you can use the Add/Edit Macros dialog box. It is not possible to create macros using CLI commands on the command line.

Complete the following steps to create a macro for use with a debug target:

1. Connect to your target and load an image, for example `dhrystone.axf`.
2. Select **Edit** → **Advanced** → **Show Line Numbers** to display line numbers.
This is not necessary but might help you to follow the examples.
3. Select **Tools** → **Add/Edit Debugger Macros...** 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.

4. 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.
5. Edit the default macro name so that it shows `int tutorial(var1)`.
6. Enter the macro contents to show:

```
int var1;
{
$printf "value=%d\n",var1$;
}
```

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.

The Add/Edit Macros dialog looks like the example shown in Figure 10-1.

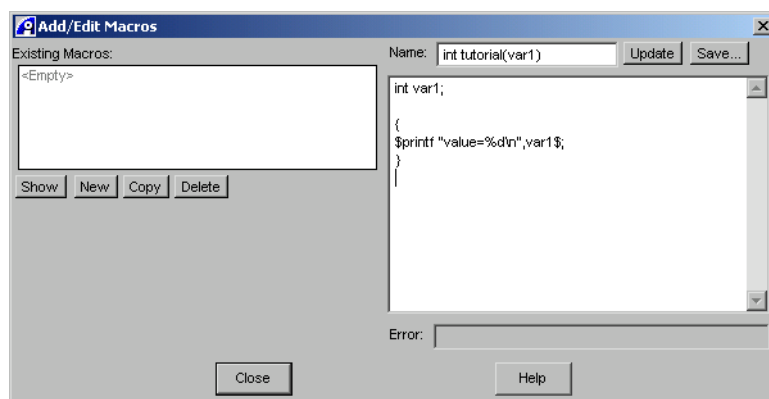


Figure 10-1 Creating a macro

The macro uses the PRINTF command and so the command must be enclosed by dollar signs (\$), shown in the Macro Entry Area.

7. Click **Update** to pass the macro definition to RealView Debugger, where it is stored 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 you try to pass the macro to RealView Debugger.

———— **Note** ————

If the symbol table is recreated, for example when an image is loaded with symbols or if you disconnect, any macros are automatically deleted.

8. View the Output pane message:

```
def int tutorial(var1)
```

9. Click **Save...** to display the Select file to save/append into dialog box where you can choose where to save the new macro definition file for later re-use.

The macro name has been entered automatically with the extension .inc.

10. Save the new macro, with the default name, in your \home directory.

If the chosen file already exists, RealView Debugger displays a warning message and gives you the option to append the new file or to overwrite the existing contents.

11. Click **Close** to close the Add/Edit Macros dialog box and return to the Code window.

10.2.2 Viewing a macro

View the contents of a macro using:

- the File Editor pane
- an external text editor
- the `SHOW` command.

Viewing a macro using any of these methods differs from 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).
- The `SHOW` command does not display the terminator.

Note

If you are using a predefined macro, you cannot use the `SHOW` command to view the definition.

10.2.3 Testing a macro

To test the macro you have created, you execute the loaded image and then enter the macro on the RealView Debugger command line:

1. Click on the **Src** tab to view the source file `dhry_1.c`.
2. Set a default breakpoint by double-clicking on line 187.
3. Click **Go** to start execution. When asked for the number of runs, use a small number, for example `5000`.
4. When the program reaches the breakpoint, enter the following command and press Enter:

```
tutorial(Int_2_Loc)
```

This displays the current value of the variable `Int_2_Loc` in the Output pane.

5. Step through the program a few more times using the macro to monitor the variable. You can use the up arrow to step back through the commands already submitted on the command line.

10.2.4 Editing a macro

To edit the macro that you have created and retest it to verify the changes:

1. Select **Tools** → **Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog. The Existing Macros display list shows the tutorial macro you created and it is highlighted.
2. Click **Show** to see the contents of the macro.
3. Change the macro name to read:

```
int tutorial(var1,var2,var3)
```

4. In the Macro Entry Area change the variable definition to read:

```
int var1;
int var2;
int var3;
```

5. In the Macro Entry Area change the body of the macro to read:

```
var1==var1++;$fprintf 250, "value=%d\n",var1$;
$fprintf 250, "value=%d\n",var2$;$fprintf 250, "value=%d\n",var3$;
```

This change increments var1 and displays the output of the macro in a window instead of the Output pane.

6. In the Macro Entry Area add this line at the end of the macro:

```
return(var1);
```

This change causes the macro to return the value of the variable. If the value is True, that is nonzero, then RealView Debugger continues program execution after reporting the result. If the value returned is False, that is zero, then execution stops.

The macro now looks like the one shown in Figure 10-2 on page 10-12.

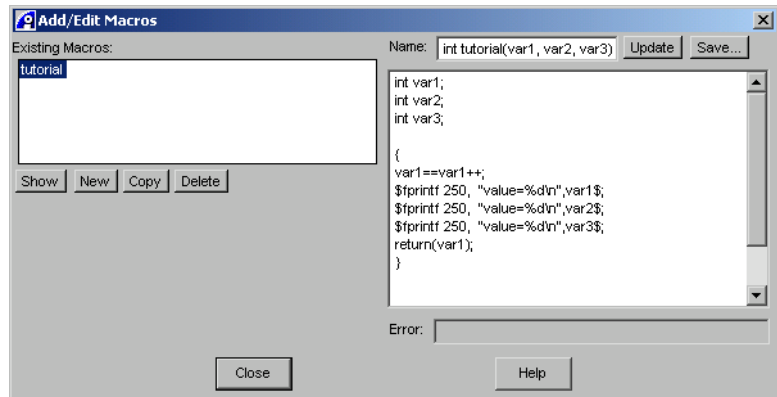


Figure 10-2 Editing a macro body

7. Click **Update** to pass the macro definition to RealView Debugger.
8. View the Output pane message:


```
def int tutorial(var1, var2, var3)
```
9. 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. Click **No** to replace the existing `tutorial.inc`.
10. Click **Close** to close the Add/Edit Macros dialog.

To test the macro you have created, you execute the image and then enter the macro on the RealView Debugger command line:

1. Select **Target** → **Reload Image to Target** to reload the image to your debug target.
2. Enter this command:


```
VOPEN 250
```

 This opens a window ready to display the results returned from the macro.
3. If you no breakpoint is set on line 187, double-click on line 187 to set a default breakpoint.
4. Click **Run** to start execution. When asked for the number of runs, use a small number, for example `5000`.
5. When the program reaches the breakpoint, enter the following command on the command line of the Code window and press the Enter key:

```
tutorial(Int_1_Loc, Int_2_Loc, Int_3_Loc)
```

This displays the current value of the variables in the window.

6. Step through the program a few more times using the macro to monitor the variables. You can use the up arrow to step back through the commands already submitted on the command line.

Remember to save your macros before you exit RealView Debugger. There is no warning if any macro definitions are unsaved.

10.2.5 Copying a macro

You can use an existing macro to form the basis of a new macro:

1. Select **Tools** → **Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog and so edit the macro. The Existing Macros display list shows the tutorial macro you created and it is highlighted.
2. Click **Show** to see the contents of the macro.
3. Click **Copy**. This automatically adds an integer number to the end of the macro name in the Name field, starting at one. For example, int tutorial1(var1,var2,var3). Subsequent copies are numbered sequentially, for example, int tutorial2... and int tutorial3.... However, you can change the this name to your own choice.
4. In the Macro Entry Area change the body of the macro as required.
5. Click **Update** to pass the macro definition to RealView Debugger.
6. View the Output pane message, assuming that you do not change the default name:

```
def int tutorial1(var1,var2,var3)
```

7. Click **Save** to save the updated macro in the usual way.
8. Click **Close** to close the Add/Edit Macros dialog.

The number of macros that can be defined is limited only by the available memory on your workstation.

10.2.6 Deleting a macro

With a group of macros shown in the Existing Macros display list, you can highlight selected macros and click **Delete** to unload them from RealView Debugger. This does not delete the files themselves if they have been saved to disk. You can only delete one macro at a time in this way.

You can also delete a macro, and all associated symbols, using the DELETE command.

10.2.7 Calling a macro

When you first start RealView Debugger, any macros you created have been unloaded from RealView Debugger.

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.

———— **Note** ————

Reloading an image with all associated symbols also deletes any macros.

To load, or reload, macros into RealView Debugger:

1. Select **Tools** from the main menu to display the **Tools** menu.
2. Select **Include Commands from File...** to display the Select File to Include Commands from dialog box.
3. 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 pane and the macro is undefined.
4. Select **Tools → Add/Edit Debugger Macros...** to display the Add/Edit Macros dialog box where your macro is now shown in the Existing Macros display list.

You can load in several macros in this way ready for use in your debugging session. When a macro is displayed in the Add/Edit Macros dialog box, it can be changed as described previously and re-used, and resaved if required.

Loading macros on connection

You can load one or more macros automatically when you connect to a given target. Do this by specifying the macros to be loaded using the Connection Properties window. Enter the calling command using the Commands setting in the Advanced_Information block, shown in Figure 10-3.

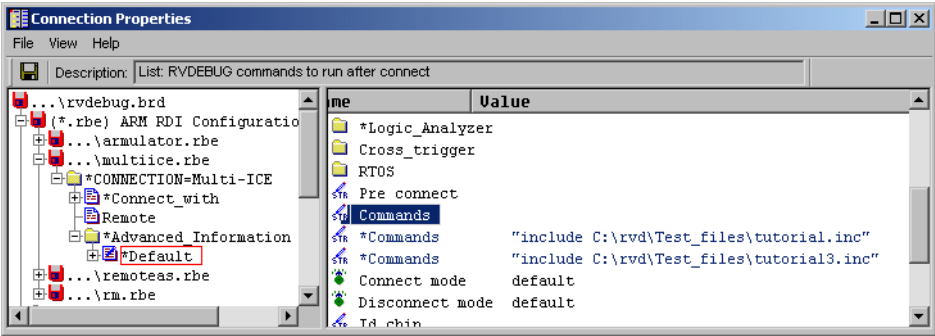


Figure 10-3 Loading macros on connection

If RealView Debugger cannot locate one of the specified files, it displays a message box and gives you the option to abort the connection.

For full details on how to specify these commands using the Connection Properties window, see Chapter 14 *Configuring Custom Connections*.

Loading macros from a project

You can load one or more macros automatically when you open a project that binds to a connection. Do this by specifying the macros to be loaded using the Project Properties window. Enter the calling command using the Open_conn setting in the Command_Open_Close group, shown in Figure 10-4 on page 10-16.

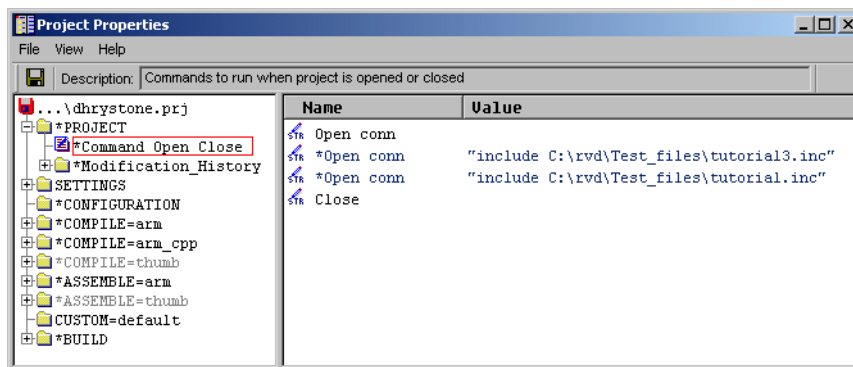


Figure 10-4 Loading macros from a project

If RealView Debugger cannot locate one of the specified files, it displays a message box. The project then opens and binds as normal.

This might be useful if combined with the `Open_load` setting in the `SETTINGS` group, for example to load the associated image and all symbols when the project binds.

For full details on how to specify these commands using the Project Properties window, see the chapter that describes how to customize your projects in *RealView Developer Kit v2.2 Project Management User Guide*.

10.3 Getting more information

See *RealView Developer Kit v2.2 Command Line Reference* for more information on how to use macros:

- see the chapter that describes working with the CLI for details on writing your own scripts
- see the chapter that describes RealView Debugger commands for full details on using the `DEFINE` command
- see the chapter that describes predefined macros for full details on predefined and user interaction macros in RealView Debugger.

Chapter 11

Configuring Workspace Settings

This chapter explains how to use workspaces in RealView® Debugger, and describes how to configure your workspace settings. Read this chapter in conjunction with Appendix A *Workspace Settings Reference* that contains a detailed description of the workspace settings.

This chapter contains the following sections:

- *Using workspaces* on page 11-2
- *Viewing workspace settings* on page 11-8
- *Configuring workspace settings* on page 11-14.

11.1 Using workspaces

RealView Debugger uses a workspace to define:

- connection information
- a list of projects to open when the next session starts
- debugger behavior
- windows (sizes and positions) and their attachment
- window contents and panes
- user-defined editor settings and view options.

It is not compulsory to use a workspace when working with RealView Debugger. However, using a workspace enables you to maintain persistence between debugging sessions.

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.

This section describes workspaces in RealView Debugger. It includes:

- *Initializing the workspace*
- *Workspace menu* on page 11-3
- *Opening workspaces* on page 11-5
- *Closing workspaces* on page 11-5
- *Projects in workspaces* on page 11-6
- *Creating an empty workspace* on page 11-7.

11.1.1 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.

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:\ARM\RVD\home\my_user_name\myws_rvdebug.aws"
```

You can start a debugging session without a workspace, for example:

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

11.1.2 Workspace menu

In a debugging session you can:

- edit your workspace settings and resave them ready for the next session
- set up specific workspaces containing custom settings for use during selected debugging sessions or with particular application programs
- switch between workspaces, without exiting the debugger, to continue previous debugging sessions
- close the current workspace and continue the session without a workspace.

To manage your current workspace, select **File** → **Workspace** from the Code window menu to display the **Workspace** menu:

Open Workspace...

Displays a dialog box where you can locate a workspace to open, see *Opening workspaces* on page 11-5 for details.

If you are already using a workspace, select this option to close the current workspace before the new workspace opens, see *Closing workspaces* on page 11-5 for details.

Save Workspace

Saves the current workspace to disk. This is useful if you have made changes since your debugging session began. The workspace file, for example `rvdebug.aws`, is saved with the same name and the workspace backup file is updated.

Save As Workspace...

Saves the current workspace to disk using a new name. This is useful if you have made changes since your debugging session began and want to save this new setup in a new workspace. This displays a dialog box where you can specify the new filename, for example `test_workspace.aws`.

The newly-specified workspace becomes the current workspace.

Workspace Options...

Displays the Workspace Options window where you can view the current workspace settings or make changes. See *Viewing workspace settings* on page 11-8 and *Configuring workspace settings* on page 11-14 for more details.

Save Settings on Exit

Selected by default, this option saves selected user-configured settings and the current workspace when you exit RealView Debugger. This enables you to start your next debugging session in the same state.

If selected, settings are saved in your start-up file, using the `.sav` extension by default, for use next time and the current workspace file is updated.

Same Workspace on Startup

Selected by default, this option saves the current workspace pathname in your `.sav` file so that the same workspace is used at the next start-up.

You can unselect 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.

Close Workspace

Closes the current workspace. After the workspace closes, RealView Debugger displays a list box so you can close any open objects. See *Closing workspaces* on page 11-5 for more details.

Recent Workspaces

Displays the Recent Workspace menu that lists any previous workspaces that you have saved. Select a workspace from this menu to use those workspace setting.

11.1.3 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 open windows before opening the new workspace, see *Closing workspaces* for details.

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.

11.1.4 Closing workspaces

You can explicitly close your current workspace, either:

- select **File** → **Workspace** → **Close Workspace**
- select **File** → **Workspace** → **Open Workspace...** to close the current workspace before the new one opens.

If you close your current workspace, the following applies:

- The contents of the default Code window do not change.
- If there are open objects, these do not change (see *Closing objects* for details).
- 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.

Closing objects

If you close a workspace, RealView Debugger enables you to close any open objects, which might be useful to restore a clean desktop for the session, or before you open a new workspace. After the current workspace closes, RealView Debugger displays a list selection box where you can specify the open objects you want to close, shown in Figure 11-1 on page 11-6.

Note

The Close Open Objects selection box is not displayed if there are no open objects.

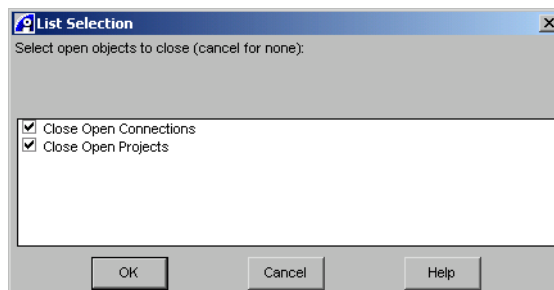


Figure 11-1 Close Open Objects selection box

The display list shows the open objects, that is:

- connections to debug targets
- any Code windows open in addition to the default Code window
- open projects including user-defined projects and auto-projects (see *Projects in workspaces* for details).

Each entry has an associated check box that is ticked by default. Select the check box to unselect objects. The list selection box contains the controls:

OK Click this button to close selected objects and then close the selection box.

Cancel Click this button to ignore the status of any check boxes in the list and close the selection box. Use **Cancel** to maintain all open objects.

Help Click this button to display the online help.

You can use the **Close** icon to close the selection box. This is the same as clicking **Cancel**.

11.1.5 Projects in workspaces

RealView Debugger saves a project load list when the current workspace closes. This is a list of open projects maintained when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

Where you are using a project load list, you must be aware of the following when the workspace opens:

- Where a project was bound to a connection when the workspace closed, binding details are saved and this is maintained when the connection is restored.
- Project binding details saved in the workspace take precedence. This applies even where the load list opens an *autobound* project that is unbound, that is where you have defined a `Specific_device` setting.
- Where there is no connection, the order in which projects open defines the active project.

If you are already running a debugging session and you open a new workspace, the project environment might change depending on the project load list (if any). RealView Debugger forces the project binding as defined in the workspace even if this means unbinding open projects. This is the case even if the open projects include an autobound project, that is a project where you have defined a `Specific_device` setting.

If the workspace opens with no saved binding details, the current project environment does not change. This is the case even if the project load list contains a project where you have defined a `Specific_device` setting.

————— **Note** —————

There is no warning when the project environment changes as a result of opening a workspace into a debugging session.

11.1.6 Creating an empty workspace

You can create a blank workspace settings file at any point during your debugging session. To do this, select **File** → **Workspace** → **New Workspace...** from the Code window main menu. This displays the New Workspace dialog box.

Use this to create an empty file, for example `New_workspace.aws`, in the new location. This becomes the current workspace. You must save settings to this new workspace settings file if you want it to be available at the next start-up.

If you are already working with a workspace this closes and then the new workspace opens ready for you to use. This does not override the current configuration.

11.2 Viewing workspace settings

RealView Debugger provides the Workspace Options window to enable you to examine, and change, workspace settings. Use this interface to see the contents of the .aws file and the .ini file.

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.

This section contains:

- *Using Settings windows*
- *Options window*
- *Workspace Options window* on page 11-9
- *Groups and settings* on page 11-12.

11.2.1 Using Settings windows

To access your current workspace settings or global configuration options:

File → Workspace → Workspace Options...

Displays the Workspace Options window where you can view the current workspace settings or make changes.

Tools → Options...

Displays the Options window where you can view the global configuration options, used by the current workspace, or make changes.

You have access to the global options when you are working without a workspace.

11.2.2 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.

If you are working without a workspace, use this window to make the changes described in the rest of this chapter.

11.2.3 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, shown in Figure 11-2.

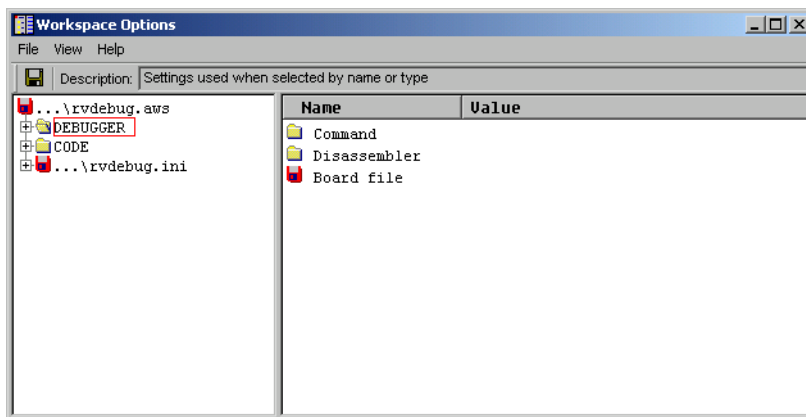


Figure 11-2 Workspace Options window

The main interface components of this window are:

Main menu This contains:

- File** Displays the **File** menu where you can save the workspace file after you have made changes.
- View** Displays the **View** menu to toggle the display to show all the settings or only those that have been edited.
- Help** Displays the online **Help** menu.

Save icon Click this icon to save the workspace settings file to disk. The name of the current file is shown as the first entry in the left pane.

Description This field displays a one-line description about an entry selected in the List of Entries pane or Settings Values pane.

List of Entries pane

The left pane is the List of Entries pane, and shows configuration entries as a hierarchical tree with node controls (see *List of Entries pane* on page 11-10).

Settings Values pane

The right pane is the Settings Values pane, and shows the settings for the group selected in the List of Entries pane (see *Settings Values pane* on page 11-11).

When you close down RealView Debugger, your workspace settings file is updated with the current configuration, for example projects, connections, and open windows.

Note

See the RealView Debugger online help topic *Changing Settings* for details on all the entries in the Workspace Options window.

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 (see Figure 11-3 on page 11-11). 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 unselected, the pencil disappears (see Figure 11-3 on page 11-11).
This icon only appears in the left 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. See Figure 11-3 on page 11-11 for an example.

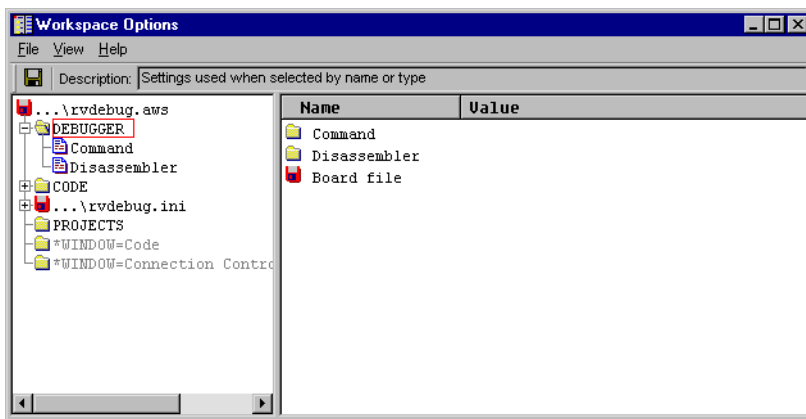


Figure 11-3 Example Workspace Options

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 *Settings Values pane*).

Settings Values pane

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. 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.

11.2.4 Groups and settings

Workspace settings, in the left pane, are grouped according to their function:

Workspace file

This is the current workspace settings file. Click on this entry to see the full pathname in the Description field.

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.

DEBUGGER

These settings govern the behavior of generic actions in the debugger. These controls are used in conjunction with other processor-specific controls.

CODE

These settings govern the behavior of all Code windows. They control the display characteristics of windows, including size and position, and any user-defined buttons created on the toolbars (not available in this release).

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.

PROJECTS

This specifies a project load list, that is a project, or projects, opened when the debugger starts with this workspace or when you open this workspace in a session. This list includes user-defined projects and any auto-projects where you have saved the settings.

This group is created automatically when RealView Debugger closes down with open projects, or if you close the current workspace.

WINDOW This is a special group of windows internals maintained by RealView Debugger. An entry is created for each open window. Entries cannot be edited.

You must not delete these entries.

CONNECT When you close down, 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.

You must not delete these entries.

For a full description of all the workspace settings you can change see Appendix A *Workspace Settings Reference*.

11.3 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.

This section describes:

- *Restoring settings*
- *Changing settings* on page 11-15
- *Copying entries* on page 11-16
- *Pasting entries* on page 11-16
- *Cutting entries* on page 11-18
- *Resetting entries* on page 11-18.

11.3.1 Restoring settings

If you have made changes to your workspace settings file, or your global configuration file, and you want to restore the factory settings:

1. Exit RealView Debugger in the normal way.
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.

11.3.2 Changing settings

This section includes examples of changes that you can make to your current workspace. Select **File** → **Workspace** → **Workspace Options...** to display the Workspace Options window to edit the workspace file. This means that changes take effect in the current workspace:

- *Configuring the Code window*
- *Setting up debugger options.*

When you have saved a change, select **View** → **New Code Window** to display a new Code window to see the effect.

Configuring the Code window

To change the size of the Code window:

1. Expand the CODE group.
2. Expand the Pos_size 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 0x040.
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**.

Setting up debugger options

To change the height of the Output pane:

1. Expand the DEBUGGER group.
2. Expand 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**.

11.3.3 Copying entries

When you are working in the Workspace Options window, context menus include options to enable you to copy settings so that you can make changes quickly. The options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.

The available options are:

Copy Select this option to make a copy of an entry to the clipboard ready for pasting.
This option is always available in the left pane to copy settings groups. When you are working in the right pane, this option depends on the entry under the cursor.

Make Copy...

Where permitted, this option enables you to make a copy of the chosen group. A dialog box enables you to define a new name for the copy.

11.3.4 Pasting entries

When you are working in the Workspace Options window, context menus include options to enable you to paste settings so that you can make changes quickly. Like the copy options, the paste options that are available depend on the:

- pane you are working in
- contents of the clipboard
- relationship between what is on the clipboard and the entry under the cursor when you right-click.

The available options are:

Paste Group Into

This option is only available if you right-click on a settings group, or a container disk file, with a settings group already copied to the clipboard.

This option is usually available in the left pane to paste settings groups into the settings file, or to copy between the workspace settings file and the global configuration file.

When you are working in the right pane, this option depends on the entry under the cursor. This means that you might not be able to paste the contents of the clipboard into the chosen location.

Paste Rule Here

Certain settings in the right pane are classed as *rules*. In particular, you can use rules to specify settings for projects when you are working in the Project Properties window. See the chapter that describes customizing projects in *RealView Developer Kit v2.2 Project Management User Guide* for details.

This option is only available if you right-click on a settings group, or a container disk file, with a single rule setting already copied to the clipboard.

This option is available in the left or right pane if you select a settings group that can accept the rule currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

Paste as 1st Child

To see this option, you must right-click on a parent group with a child group already copied to the clipboard.

Use this option, in the left pane, to paste a settings group into a parent group, that is to create a sibling group.

When you are working in the right pane, use this option to paste a child group. The paste only succeeds if the chosen parent group can accept the child. This means that you might not be able to paste the contents of the clipboard into the chosen location.

This option might be replaced by **Paste After**. This depends on the entry under the cursor.

Paste After Use this option, like **Paste as 1st Child**, to paste a settings group into a parent group, that is to create a sibling group.

This option enables you to specify the relationship between the new group and other siblings. This determines the order in which settings are used.

Paste String This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a string setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the string currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

Paste Value This option is only available in the right pane if you right-click on a settings group, or a container disk file, with a value setting already copied to the clipboard.

The paste only succeeds if you select a setting that can accept the value currently on the clipboard. This means that you might not be able to paste the contents of the clipboard into the chosen location.

11.3.5 Cutting entries

When you are working in the Workspace Options window, context menus include the option **Cut** to enable you to mark an entry for deletion. The entry is grayed out until you paste it into a new location.

If you cut another entry before you paste the first entry, the first entry is restored.

If you cut an entry, do not delete it until it has been pasted. If you delete the cut entry, it is no longer available to paste. This also applies to entries that you copy.

Always use a **Delete** option to remove unwanted entries.

11.3.6 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. You can reset all values using the **File** → **Reset** option from the window 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.

Chapter 12

Connection and Target Configuration

This chapter introduces the connection and target configuration system used by RealView® Debugger. It contains the following sections:

- *About target connections and configuration* on page 12-2
- *Viewing board file properties* on page 12-5
- *Configuration files* on page 12-11.

Note

RealView Developer Kit can only connect to particular ARM® architecture-based processors; each version of RVDK is designed to be used only with a particular device, or range of devices. See your *RealView Developer Kit v2.2 Getting Started Guide* for information on permitted devices.

The debugger allows connection to the RealView ICE, RealView Trace and RealView ICE Micro Edition supplied with RVDK.

12.1 About target connections and configuration

RealView Debugger uses a *board file* to access information about the debugging environment and the debug targets available. The board file, therefore, describes how RealView Debugger connects to, and interacts with, your debug targets.

The default board file is set up when you first install RealView Debugger. It is called `rvdebug.brd` and, along with files that it references, is stored in your default home directory. For details on your home directory, and what it contains, see *The home directory* on page 12-13.

This section describes target connection and configuration, and introduces the features of RealView Debugger that enable you to define these settings:

- *Comparing target connection and target configuration*
- *Connection Control window* on page 12-3
- *Connection Properties window* on page 12-4.

12.1.1 Comparing target connection and target configuration

RealView Debugger makes a distinction between:

Target connection

Describes how the debugger accesses your debug target.

Target configuration

Describes your debug target to the debugger, for example how memory is mapped.

Specifically, the board file enables you to specify high-level connection details such as:

- debugger to target connection details, such as interface type and instance, TAP controller positions, and connection interface address
- debugger actions taken when a connection is made, for example running commands and opening projects.

And low-level, target configuration details such as:

- target processor characteristics, for example processor type and endianness
- target peripheral register and memory configuration.

To do this, the default board file contains a series of elements arranged in a hierarchy. At the top level are connection entries or *target vehicles* that enable RealView Debugger to connect to different targets. As you drill down the connection, you can modify entries, or create new elements, that define the target configuration in more detail.

At the lowest level, RealView Debugger is able to access *Extended Target Visibility* (ETV) information about your debug target using a special group of settings, the `Advanced_Information` block that is found in all the main groups.

RealView Debugger can also access Board/Chip definition files that contain ETV information about a particular board or chip as supplied by the manufacturer, including peripheral registers and memory regions.

12.1.2 Connection Control window

You can use the Connection Control window to make your first connection to a debug target. The RealView Debugger base product includes built-in configuration files to enable you to make a connection without having to modify any configuration details.

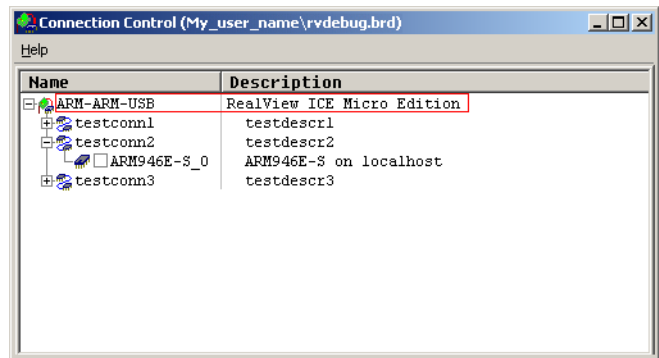


Figure 12-1 Connection Control window

12.1.3 Connection Properties window

The contents of the Connection Control window are defined by elements of the board file. To change target configuration, or to add new debug targets, you change the entries in the board file and this modifies the elements displayed in the Connection Control window. RealView Debugger provides a GUI interface, the Connection Properties window, to make these changes. See *Connection Properties window* on page 12-5 for more details.

———— **Note** —————

Do not configure the board file when the debugger is connected to a target.

The rest of this chapter describes RealView Debugger target configuration in more detail.

Getting more information

Appendix B *Configuration Properties Reference* provides a detailed reference that describes all the configuration settings available in the board file. It provides examples of when to use settings and describes how to make changes. This might be useful when you are working through the examples in this chapter and in the rest of this book.

12.2 Viewing board file properties

RealView Debugger provides the Connection Properties window to enable you to examine, and change, target connections and configuration details stored in the board file. This section describes these entries in more detail and contains the following sections:

- *Connection Properties window*
- *Connection entries on page 12-8*
- *Configuration entries on page 12-9.*

12.2.1 Connection Properties window

Select **Target** → **Connection Properties...** to display the Connection Properties window, shown in Figure 12-2. This enables you to examine your current configuration settings and edit these settings to set up new connections and change target configuration.

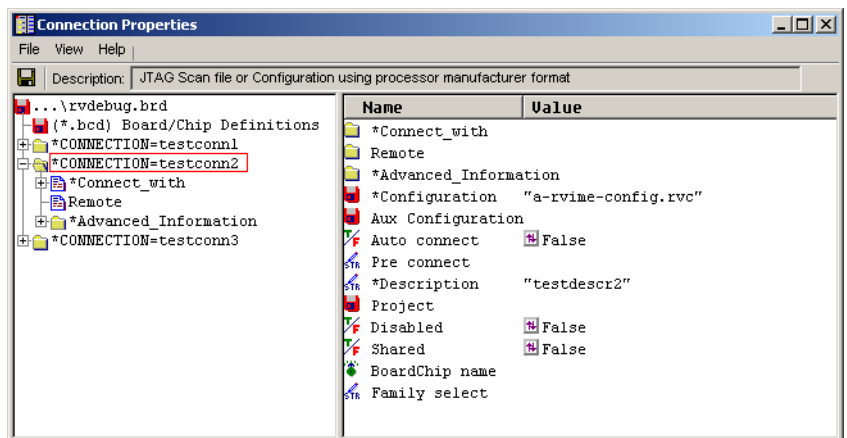


Figure 12-2 Connection Properties window

Note

Do not make changes in the Connection Properties window when the debugger is connected to a target.

Connection Properties window interface components

The main interface components of the Connection Properties window are:

Main menu This contains:

- File** Displays the **File** menu where you can save the board file after you have made changes.
- View** Displays the **View** menu to toggle the display to show all the settings or only those that have been edited.
- Help** Displays the online **Help** menu.

Toolbar The toolbar contains:

Save icon

Click this icon to save the configuration settings file to disk. The name of the current file is shown as the first entry in the left pane.

Description

This field displays a one-line description about an entry selected in the panes below.

List of Entries pane

The left pane is the List of Entries pane, and shows configuration entries as a hierarchical tree with node controls (see *List of Entries pane* on page 11-10).

Settings Values pane

The right pane is the Settings Values pane, and shows the settings for the group selected in the List of Entries pane (see *Settings Values pane* on page 11-11).

Note

See the RealView Debugger online help topic *Changing Settings* for details on all the entries in the Connection Properties window.

RealView Debugger uses the board file to configure the target and the connection with two types of entry:

- *Connection entries* on page 12-8
- *Configuration entries* on page 12-9.

List of Entries pane

The left pane of the Connection Properties window, the List of Entries pane, shows configuration entries as a hierarchical tree with node controls. 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. This icon only appears in the left pane.

An asterisk (*) is placed at the front of an entry to show that it has changed from the default or that it was changed by RealView Debugger.

Settings Values pane

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. 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 target device configuration 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.

12.2.2 Connection entries

The Connection Control window, shown in Figure 12-1 on page 12-3, uses the connection entries in the board file to create a tree of possible connections. The entries in the board file reference device configuration files (such as .jtg and .rvc) that specifies the processors that can be reached using that connection, for example ARM940T_0.

To see these connection entries you must install other ARM products because RealView Debugger does not include any connection software of its own. However, the debugger does include connection interface components, or target vehicles, to provide the interface to each target. For example, as shown in Figure 12-1 on page 12-3, the ARM-ARM-USB vehicle provides the interface to RVI-ME targets.

Figure 12-1 on page 12-3 shows the Connection Control window generated by the board file entries shown in Figure 12-2 on page 12-5. Each connection entry in the board file has a corresponding entry in the Connection Control window.

You can create your own target connection groups, for example CONNECTION=testconn3 shown in Figure 12-2 on page 12-5. This new connection appears as an entry in the Connection Control window, that is testconn3, shown in Figure 12-1 on page 12-3. You can create new groups based on existing connection groups or set these up using RealView Debugger defaults. You can also add any groups that are missing when you run RealView Debugger. See Chapter 14 *Configuring Custom Connections* for details on how to do this.

———— Note ————

For more details on the entries in the Connection Control window, see *Working with the Connection Control window* on page 2-2.

12.2.3 Configuration entries

Target configuration entries enable you to describe the target architecture to RealView Debugger. This makes it possible for the debugger to present peripheral registers in a more human-readable format, and enables operations involving target memory to take account of the target memory map, for example so that Flash memory can be accessed correctly.

Target configuration information, using BOARD, CHIP, and COMPONENT groups, is used to define a hierarchy, starting from the general board-level and becoming more specific, through whole chips to component modules on a chip. However, RealView Debugger does not distinguish, functionally, between the different group names and you can use them as you require.

Using .bcd files

Within the top-level board file, rvdebug.brd, you can have as many BOARD, CHIP, or COMPONENT entries as you require. However, there is a better way to store them.

When RealView Debugger starts up, it searches for files with the extension bcd and loads them into a group called (*.bcd) Board/Chip Definitions, shown in Figure 12-3.

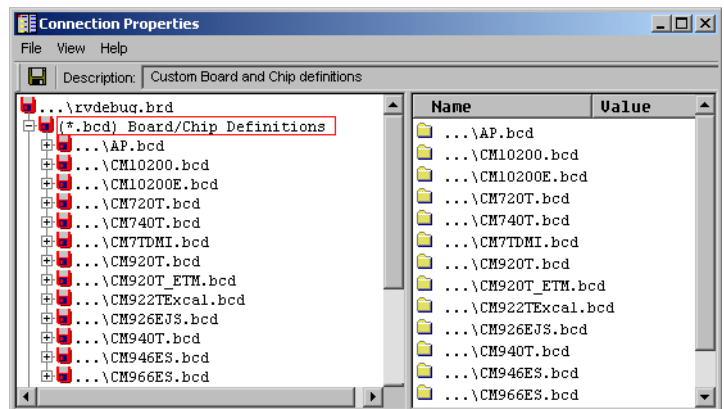


Figure 12-3 Viewing .bcd files in the Connection Properties window

Configuration entries in files loaded into this group can be referenced from any connection. For example, you can define a connection that references the file AP.bcd, to access the ARM Integrator/AP board, and the file CM940T.bcd, to access the ARM940T processor core module registers and memory map. In this way, the connection defines a hierarchy of configuration details from the board level to the processor level.

This makes the target description independent of the connection used to access it, and makes it easier to reuse target descriptions in different debugging sessions.

Using the Advanced_Information block

Extended target visibility is possible using a special group, the Advanced_Information block that is found in almost all the main groups, shown in Figure 12-2 on page 12-5. For example, there is an Advanced_Information block in the CONNECTION=testconn2 entry of the ARM-ARM-USB group, and in the BOARD=AP entry of the (*.bcd) Board/Chip Definitions group.

Note

In the board file, both target connection groups, for example CONNECTION=testconn2, and target configuration groups, for example BOARD=AP, contain an Advanced_Information block. Although you can use the Advanced_Information block of the target connection group, it is suggested that you use target configuration groups and then reference these from the connection entry you are using. To ensure that these settings are used, do not change the default settings contained in the target connection group.

The search procedure, the way files are referenced, and the configuration options are described in more detail in Chapter 13 *Configuring Custom Targets*.

12.3 Configuration files

Default configuration files are supplied as part of the RealView Debugger base product that defines standard ARM targets.

12.3.1 What the configuration files contain

The configuration files that RealView Debugger stores in your home directory include the following files relating to this book:

- *.brd Top-level board files. Normally this is `rvdebug.brd`. This file includes the filenames of the other configuration files, for example `.bcd` and `.jtg` files.
You might change this file when you save settings in the Connection Properties window (see the examples in *Creating new target descriptions* on page 13-14).
- *.bcd Board/Chip definition files supplied by hardware manufacturers.
Files that contain configuration information for specific targets. By default, these files are used from the default settings directory, `\etc`.
If you change a supplied file or you create your own, you are recommended to store them in your debugger home directory.
These files contain per-chip and per-board settings as named configurations.
You might change this file when you save settings in the Connection Properties window (see the examples in *Creating new target descriptions* on page 13-14).
- *.prc These files are used if you install new products such as RVI and RVI-ME. For example, the first time RealView Debugger runs after you install RVI-ME, the information contained in these files is copied into the board file. After this happens, the `.prc` file is ignored.
- *.rvc Configuration files for RV-msg connections. You change the contents of one of these files when you modify the configuration of an RV-msg connection using the RVConfig dialog box (see the example in *Configuring an RVI/RVI-ME interface unit* on page 14-10).
For details on working with `*.rvc` files when using RealView ICE, see the chapter on configuring a RealView ICE connection.

12.3.2 Finding configuration files

This section describes how RealView Debugger directories are created when you install the base product, and how they are used to find the configuration files. It contains the following sections:

- *The install directory*
- *The home directory* on page 12-13
- *The RealView Debugger search path* on page 12-14
- *Saving and restoring connection properties* on page 12-14.

The install directory

RealView Debugger must be able to locate the product installation directory so that it can locate program extensions, data, and configuration files stored there. Use the following settings to do this:

- The default location, for example:
C:\Program Files\ARM\RVD\Core\1.8\build_number\win_32-pentium
- The RVDEBUG_INSTALL environment variable:

set RVDEBUG_INSTALL=D:\ARM\RVD\Core\1.8\build_number\win_32-pentium
For information about environment variables in RealView Debugger, see Chapter 1 *Starting to use RealView Debugger*.
- The -install command line option:

-install="D:\ARM\RVD\Core\1.8\build_number\win_32-pentium"

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

The shortcuts that are installed on the Windows **Start** → **Programs** 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.

The home directory

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

The location of the home directory depends on the environment variables and command-line options defined when the debugger is started. RealView Debugger uses the first defined item from the following ordered list:

1. On Windows, you can use the `-home` argument when starting RealView Debugger from the command line to specify an explicit path:

```
-home="C:\Documents and Settings\user_name\RVD"
```

The home directory is `C:\Documents and Settings\user_name\RVD`

2. You can use the `RVDEBUG_HOME` environment variable to use a directory other than the default home directory. For example, on Windows:

```
set RVDEBUG_HOME=C:\RVD\Test_home
```

For information about environment variables in RealView Debugger, see Chapter 1 *Starting to use RealView Debugger*.

3. You can use the `-install` command line option on its own or together with `-user` or `$USER`:

```
-install="D:\WinApp\RVD\" -user="DevTeam"
```

The home directory is `D:\WinApp\RVD\home\DevTeam\`.

You can use `-user` or `$USER` to specify an alternative for `DevTeam`.

4. You can use the `RVDEBUG_INSTALL` environment variable on its own or together with `-user` or `$USER`:

```
set RVDEBUG_INSTALL=D:\WinApp\RVD\
```

```
set USER=DevTeam
```

The home directory is `D:\WinApp\RVD\home\DevTeam`.

5. The default location, for example:

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

Note

If the debugger home directory location is defined, but the directory does not exist, the debugger creates it and copies the standard set of configuration files there from the default settings directory (\etc).

The RealView Debugger search path

RealView Debugger searches several directories for board files, including the default file, `rvdebug.brd`. The search path the debugger uses is:

1. The current working directory, sometimes called the *Start In* directory.
2. The location defined by the `RVDEBUG_SHARE` environment variable, if it is set.
For information about environment variables in RealView Debugger, see Chapter 1 *Starting to use RealView Debugger*.
3. The RealView Debugger home directory, described in *The home directory* on page 12-13, using the order specified there.
4. The default settings directory, \etc.

RealView Debugger searches all of these directories for workspace files and other configuration files. In particular, this is how `*.bcd` files are found. Where two or more files with the same filename are found in more than one of the searched directories, the first file found is loaded and others are ignored.

Saving and restoring connection properties

When you are configuring RealView Debugger, you are recommended to keep backups of known-good configuration information before changing settings. There are backup systems you can use:

- You can rely on the automatic backup that RealView Debugger makes whenever it saves the Connection Properties window, for example `rvdebug.brd.bak`.
- You can copy specific files, for example `rvdebug.brd`, or the whole home directory to a backup area.

Note

It is recommended that you make backups before using the worked examples in the rest of this book.

Using the automatic backup files

If you edit a board file, or a Board/Chip definition file, RealView Debugger automatically renames the original file by adding a .bak file extension. Any previous backup copy of the file is deleted.

If you want to restore a backup file:

1. Exit the Connection Properties window without saving changes.
2. Delete the current file or files.
3. Rename the backup file to the original filename by deleting .bak from the name.

Using manual file or directory backups

For safer backups, you are recommended to make tape or disk copies of the files in another place. The simplest policy is to save the whole directory when you make a backup, but restore individual files when you want to revert changes.

Note

If you restore the whole directory, then in addition to restoring the Connection Properties configuration information, you restore preferences that you might not want to change, for example workspace properties, project properties, and window layout.

Creating a directory backup requires you to locate and copy the home directory to a safe place. You do not have to exit the debugger to do this.

Restoring previously backed up files requires you to:

1. Locate the backup that you want to restore from and the debugger home directory that RealView Debugger is using for your session.
See *The home directory* on page 12-13 for more information about the location of your debugger home directory.
2. Determine the files to restore.
3. Copy the backup files to the debugger home directory.

Deciding which files to restore depends on the type of configuration change you have performed. These hints might help:

- If you have changed everything, or you are not sure what to restore, select all the files listed in *What the configuration files contain* on page 12-11 to restore the Connection Properties window to its original state.

- If you have configured or reconfigured a chip or board using BOARD, CHIP or COMPONENT groups, select appropriate files from the *.bcd set.
If you have created new *.bcd files in your debugger home directory, you might also want to delete them from that directory. However, a *.bcd file is not used unless it is explicitly referenced.
- If you have changed RV-msg connection settings, for example by changing items in the RVConfig dialog box, select the *.rvc files for the RVI/RVI-ME connections you have reconfigured. These files are saved by default in your RVI/RVI-ME installation directory.

Chapter 13

Configuring Custom Targets

This chapter describes in detail the debug target configuration model used by RealView® Debugger. Read this chapter to find out how to describe your debug target to the debugger. It contains the following sections:

- *About target configuration* on page 13-2
- *Linking a board, chip, or component to a connection* on page 13-6
- *Creating new target descriptions* on page 13-14
- *Example descriptions* on page 13-22.

Note

RealView Developer Kit can only connect to particular ARM® architecture-based processors; each version of RVDK is designed to be used only with a particular device, or range of devices. See your *RealView Developer Kit v2.2 Getting Started Guide* for information on permitted devices.

The debugger allows connection to the RealView ICE, RealView Trace and RealView ICE Micro Edition supplied with RVDK.

13.1 About target configuration

This section describes target configuration in more detail in readiness for the examples in the rest of this chapter. This section describes:

- *Target configuration settings*
- *Default configuration files*
- *How configuration files are linked together* on page 13-5
- *Board file contents* on page 13-5.

13.1.1 Target configuration settings

RealView Debugger assembles configuration settings to describe the debug environment and all the debug targets available in the current debugging session. These settings serve two main purposes. They:

- describe your debug targets in a way that enables RealView Debugger to find out all the information it requires to establish a connection
- enable you to configure the *Extended Target Visibility* (ETV) features of your debug targets, and make this information accessible to RealView Debugger.

Using internal configuration settings in this way means that you can change your debug target connection, or connect to multiple debug targets, without leaving your RealView Debugger session.

13.1.2 Default configuration files

As introduced in Chapter 12 *Connection and Target Configuration*, the debug target configuration settings are maintained through the use of a hierarchy of configuration files:

- *Board file* on page 13-3
- *RV-msg configuration files* on page 13-3
- *Board/Chip definition files* on page 13-4.

———— **Note** ————

Some default configuration files are supplied as part of the RealView Debugger installation, see *Configuration files* on page 12-11 for details.

Board file

RealView Debugger uses a board file to access information about the debugging environment and the debug targets that are available. You can use RealView Debugger with the default board file that is installed for you. This is called `rvdebug.brd` and is copied into your home directory, from the default settings directory `\etc`, when you first use RealView Debugger after installation. This means that if you damage your personal board file, you only have to delete it from your home directory and a new copy of the original default board file is placed there when you next run the debugger.

The board file defines the debug target configuration settings for the current session. For each available target, it describes the type of target, the simulator or emulator being used, and any custom connection information.

RealView Debugger must have a board file to make connections. If you work with a variety of targets and connections, you might set up, and save, several board files so that you can easily switch the debugger from one to another. You can use the default board file as a basis for making many copies, each edited for a particular purpose.

You can use a text editor to display or print the contents of a board file, and all associated configuration files, but it is recommended that you never edit these files with a text editor or word processor. Use only the Connection Properties window to make changes to a board file, or to create a new one.

RV-msg configuration files

These files are used to define the configuration settings for RV-msg connections, such as RVI and RVI-ME. You change the contents of one of these files when you modify the configuration of an RV-msg connection using the RVConfig dialog box.

Each interface unit is defined using a `.rvc` file, for example `rvi_940T-_tst.rvc`. However, you can specify different `.rvc` files to configure custom targets.

You can specify a full pathname in your board file to use a different location, for example `\RVI\test_targets\rvi_920T-_tst.rvc`.

Note

Do not edit these files manually. See Chapter 14 *Configuring Custom Connections* for full details on how to use the RVConfig dialog box.

Board/Chip definition files

Board/Chip definition files contain ETV information about a particular board or chip as supplied by the manufacturer, including peripheral registers and memory regions. Board/Chip definition files are also supplied as part of support plugins to enable awareness in RealView Debugger, for example RTOS.

Each board or chip is defined using a file named *filename.bcd*, where *filename* identifies the scope of the file contents, and can be:

- a processor name, for example CM920T_ETM.bcd
- a board name, for example, AP.bcd
- a peripheral name or other meaningful name, for example, realmonitor.bcd.

By default, .bcd files are stored in \etc, but you can specify a different location in your board file.

In general, you do not have to edit these files. However, where changes are required, use the Connection Properties window to make the necessary changes.

13.1.3 How configuration files are linked together

The board file might reference several other configuration files, for example the Board/Chip definition files, *.bcd., to form the complete configuration. This relationship is shown in Figure 13-1.

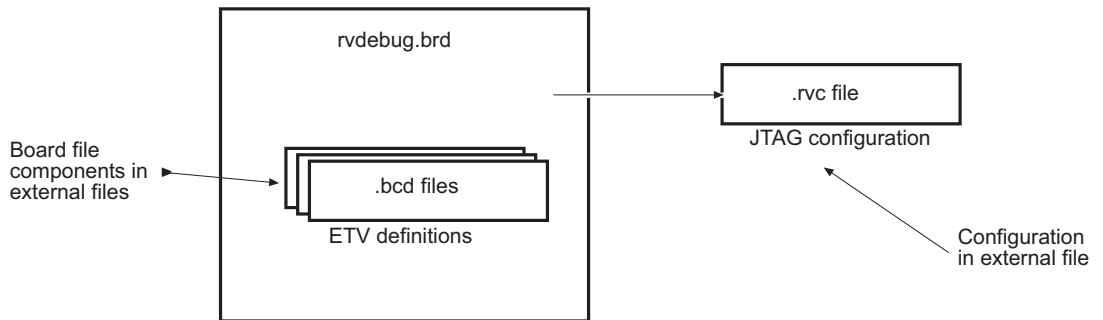


Figure 13-1 Configuration files in the board file

The RV-msg and JTAG configuration files contain the remaining information required to configure a specific debug target. These files are not structured in the same way as the board files. They use the format required by the debug target that they are used to configure.

13.1.4 Board file contents

Appendix B *Configuration Properties Reference* provides a detailed reference that describes all the configuration settings available in the board file. It provides examples of when to use settings and describes how to make changes. This might be useful when you are working through the examples in this chapter.

13.2 Linking a board, chip, or component to a connection

The target description groups in a *.bcd file are only used if:

- the .bcd file is referenced from a connection
- the identification string, `id_match`, specified in the `Advanced_Information` group, matches the hardware ID string returned by the target.

If more than one linked configuration group matches the target hardware, the group with the longest match is used.

This section describes how you make these links in the board file for a connection. These examples assume that you are linking configuration groups to a connection that uses RVI-ME. To avoid conflict between settings in different configuration groups, see *Avoiding conflicts between linked board groups*.

For details on how to configure RealView ICE connections, you should refer to the chapter on configuring a RealView ICE connection in *RealView ICE and RealView Trace Version 1.5 User Guide*.

If this is the first time you have used RVI-ME, ensure that you configure the debug interface before trying to connect (see *Configuring an RVI/RVI-ME interface unit* on page 14-10 for details).

There are several cases to consider, presented here in order of increasing complexity:

- *Linking one board group to one connection* on page 13-7
- *Linking several board groups to one connection* on page 13-9
- *Linking one or more board groups to another board group* on page 13-11

———— Note ————

The examples in this section use existing target descriptions. However, you can create and use your own target descriptions. For detailed instructions describing how to create your own target descriptions, see *Creating new target descriptions* on page 13-14.

13.2.1 Avoiding conflicts between linked board groups

When you link target description groups to a connection, make sure that you change settings in one place to avoid conflicts:

- For single processor configurations, it is suggested that you make changes at the CONNECTION level.

13.2.2 Linking one board group to one connection

This configuration is shown in Figure 13-2.

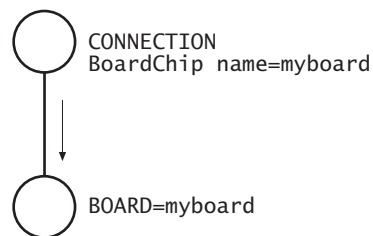


Figure 13-2 Linking one connection to one board

To set up this linking configuration you must first ensure that the *.bcd file exists and contains the required BOARD, CHIP, or COMPONENT groups. Then reference this file from your board file:

1. In the Connection Properties window, click the connection that you are using, for example CONNECTION=testconn2 shown in Figure 13-3.

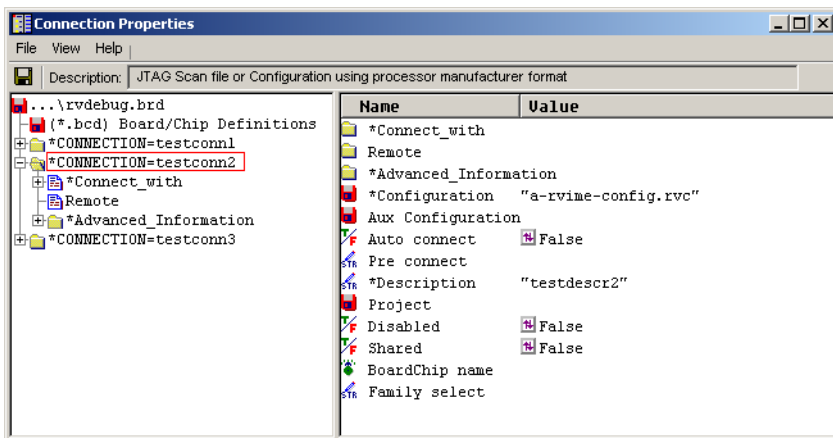


Figure 13-3 testconn2 connection properties

2. Right-click on the BoardChip_name setting, in the right pane, to display the **BoardChip** context menu shown in Figure 13-4 on page 13-8.

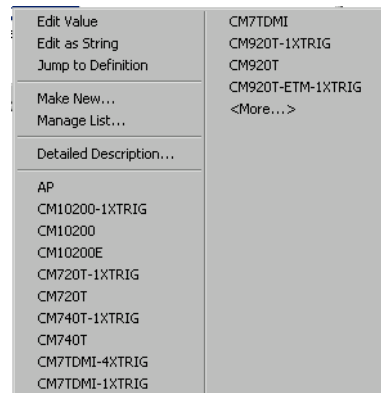


Figure 13-4 Linking a board

This context menu does not show the names of the .bcd files found by RealView Debugger. Rather, the menu includes entries derived from the contents of the .bcd files. It shows all BOARD, CHIP and COMPONENT groups in the order that RealView Debugger locates them (RealView Debugger searches your home directory first, then the ...etc directory). This illustrates how you can share target descriptions between different connections.

3. Select the name of the required group, for example **AP** for the Integrator/AP motherboard. A new entry is displayed in the right pane, which is colored blue, and has an asterisk * beside it, for example *BoardChip_name AP, shown in Figure 13-5.

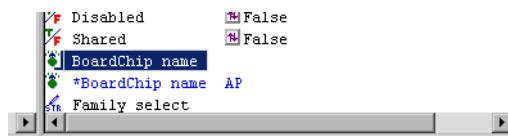


Figure 13-5 New BoardChip_name setting in connection properties

4. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

When you connect to your RVI/RVI-ME target, the configuration defined in your board group is applied to the connection. This gives you visibility of your custom hardware.

13.2.3 Linking several board groups to one connection

You might want to link several groups to a single connection if the groups represent different, possibly optional, parts of the same target. For example, the Integrator/AP motherboard definition `BOARD=AP` and an Integrator core module definition such as `BOARD=CM940T`. This kind of layout is shown in tree form in Figure 13-6.

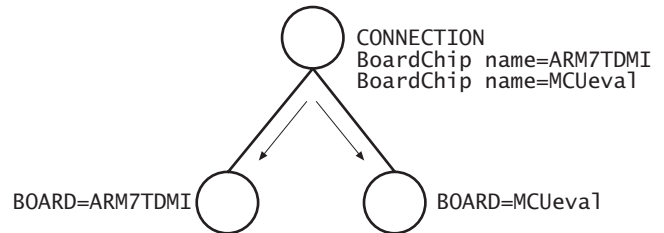


Figure 13-6 Linking one connection to two boards

When you reference multiple boards, RealView Debugger merges the settings from each matching group. Therefore the complete configuration is the combined configurations of all of the matching groups. If the same setting is specified in more than one group, the specification in the group that is listed first in the `CONNECTION` is used, for example `BoardChip_name=AP` in Figure 13-6.

To set up this linking configuration you must first ensure that the `*.bcd` files exist and contain the required `BOARD`, `CHIP`, or `COMPONENT` groups. Then reference these files from your board file:

1. In the Connection Properties window, expand the connection that you are using, for example `CONNECTION=testconn2` shown in Figure 13-3 on page 13-7.
2. Link the AP target description as described in *Linking one board group to one connection* on page 13-7, unless you have already done this.
3. Right-click on the `BoardChip_name`, in the right pane, to display the **BoardChip** context menu shown in Figure 13-4 on page 13-8.
4. Select the name of the required group, for example **CM940T**. You might have to select the **<More...>** option to find the group you require. A new entry is displayed in the right pane with an asterisk `*` beside it, shown in Figure 13-7 on page 13-10 (that is `*BoardChip_name CM940T`).

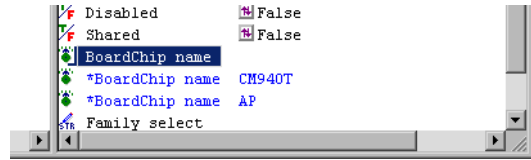


Figure 13-7 Linking a second board

5. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

When you connect to your RVI/RVI-ME target, the configuration defined in your board groups is applied to the connection. This gives you visibility of all your custom hardware.

Use the method described here to set up as many configuration groups as you require.

Changing the order of board groups

This procedure describes adding board CM940T after board AP. A board group is always added at the top of the list.

However, you might want to reorder the boards in the BoardChip_name list (to recreate the structure shown in Figure 13-6 on page 13-9). To do this:

1. Right-click on any BoardChip_name entry to display the **BoardChip** context menu and select **Manage List...**. The Settings: List Manager dialog box is displayed.
2. Select the checkbox for the AP entry, so that it is checked.
3. Click **Move Up**. The AP entry is moved above the CM940T entry.
4. Click **OK** to close the dialog box. The BoardChip_name list is reordered.
5. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

13.2.4 Linking one or more board groups to another board group

You might want to link several groups together so that you can share descriptions or simplify each part of a description. For example, the description of the ARM Evaluator-7T provided in Eval7T.bcd is split into a description of the board, *BOARD=Evaluator7T, and a description of the processor on the board, KS32C50100 or S3C4510B depending on the version of the board. This is shown in Figure 13-8.

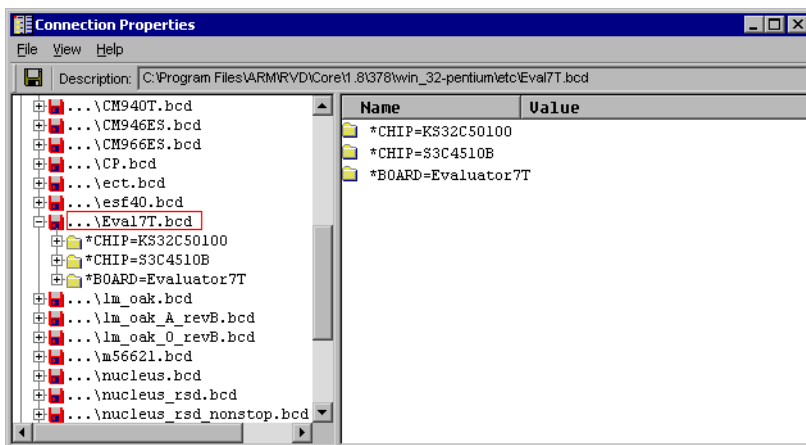


Figure 13-8 Board and chip groups for the Evaluator-7T

This is shown in tree form in Figure 13-9.

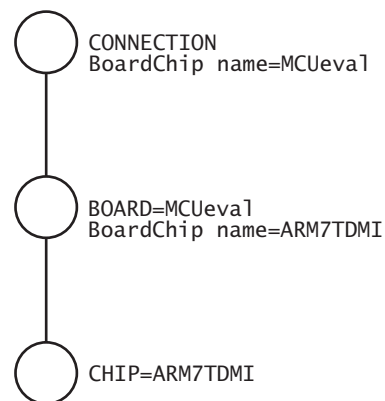


Figure 13-9 Linking one board into another board

Groups can contain BoardChip_name references to other groups, so that you can build multi-layered descriptions. For example, if you were building a simple Ethernet router, you might use the network interface on the KS32C50100 with a second network interface provided by an AMD LANCE. This configuration is shown in Figure 13-10.

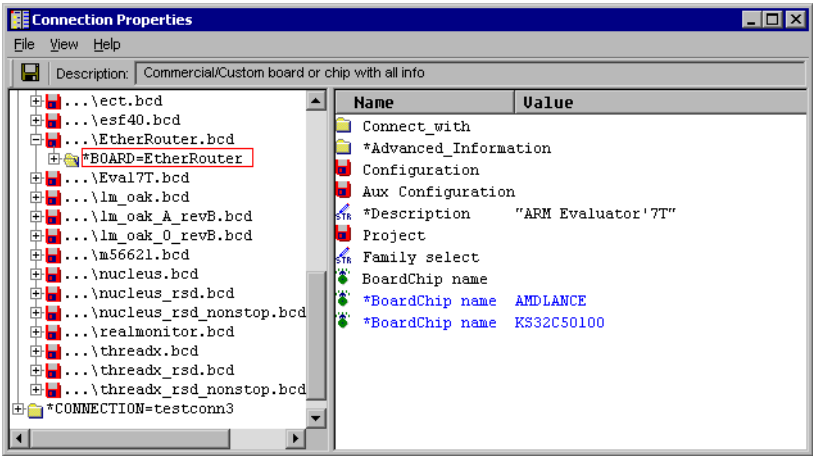


Figure 13-10 Board and chip groups for the EtherRouter

This is shown in tree from in Figure 13-11.

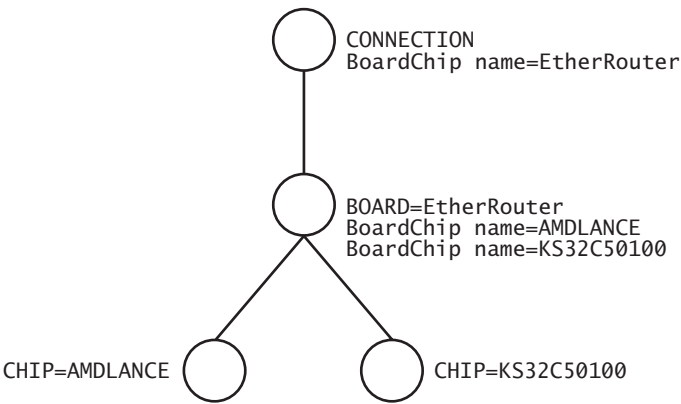


Figure 13-11 Linking one board into other boards

Note

You are not required to split your board up into distinct CHIP descriptions. You could create one BOARD description containing all of the required information. However, splitting your board up into distinct CHIP descriptions enables you to share descriptions or reuse a description for another development project.

An example showing how to create this structure is described in *Creating multiple board groups and linking to another board group* on page 13-18.

13.3 Creating new target descriptions

This section explains how to create variations of existing configurations and new target descriptions. Creating new target descriptions, rather than modifying existing entries, provides certain advantages, for example:

- if you define the addresses of I/O registers, and the bit fields within them, the debugger can display tabs in the Register pane enabling GUI access to these values
- if you define a memory map, the debugger can check that memory is used correctly, including refusing to load programs where there is no memory, and automatically invoking Flash memory programming routines.

If you want to create and use a new target description, you must:

1. Create a new `.bcd` file to store the configuration. The recommended way to do this is to base the new file on an existing file stored in the default settings directory, see *Creating a new .bcd file* on page 13-15 for details.
2. Create and name a BOARD, CHIP, or COMPONENT group for the configuration. Which group you use depends on the type of hardware you are describing.
3. Define the CONNECTION entry to specify the access method used to connect to the new target.

For examples that describe how to create new CONNECTION entries, see *Example descriptions* on page 13-22.

4. Link the new target definition to the CONNECTION group that the target uses.

The rest of this section describes how to configure the new target, that is:

- *Creating a new .bcd file* on page 13-15.
- *Creating and naming a board, chip, or component* on page 13-17.
- *Linking a board, chip, or component to a connection* on page 13-6.

For information about how these different configuration groups interact, that is configuration settings, the board file, and any linked Board/Chip definition files, see *Managing configuration settings* on page 13-20.

13.3.1 Using the examples

The examples in the rest of this chapter, modify the board file stored in your RealView Debugger home directory. By default, this is called `rvdebug.brd`. Target configuration files might also be stored in this directory, for example `.rvc` files or `.cnf` files.

It is recommended that you back up this directory before starting the examples, so that you can restore your original configuration later. For details see:

- *Configuration files* on page 12-11 for instructions on making backups of your configuration.
- *Restoring your .brd file* on page 13-46 for instructions on restoring a default configuration.
- *Troubleshooting* on page 13-46 for instructions on recovering from an incorrectly configured debugger home directory, whether or not you have a backup.

Note

Remember, when you are following these examples, do not configure the board file when the debugger is connected to a target.

Before you start, remove the linked AP and CM940T board descriptions from the connection, if you have been following the instructions in *Linking a board, chip, or component to a connection* on page 13-6. To do this right-click on each *BoardChip_name entry, and select **Delete** from the **BoardChip** context menu. Then select **File** → **Save and Close** to save the changes and close the window.

13.3.2 Creating a new .bcd file

To create a new *.bcd file, copy one of the existing files within RealView Debugger or use the tools provided by your operating system, for example, using Windows Explorer.

To copy a *.bcd file from RealView Debugger use the Connection Properties window:

1. Select **Target** → **Connection Properties...** to display the Connection Properties window.
2. Expand the group (*.bcd) Board/Chip Definitions by clicking on the plus sign. This displays the current list of target descriptions, in the left pane.
3. Right-click on the name of the *.bcd file to copy. For example, right-click on the entry ... \AP.bcd.
4. Select **Save As...** from the context menu, to display the dialog box shown in Figure 13-12 on page 13-16.

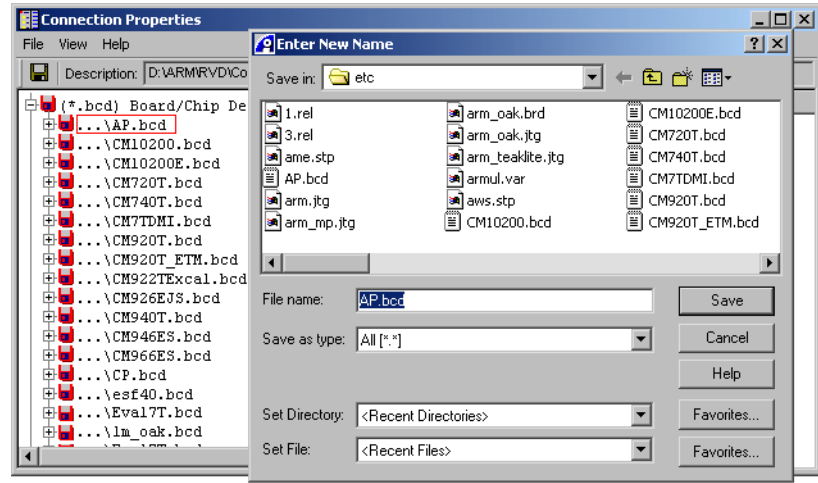


Figure 13-12 Saving an existing .bcd file with a new name

By default, *.bcd files are saved in your default settings directory \etc.

In this example, save the new file in your RealView Debugger home directory.

5. Enter the new filename, for example AM.bcd. You must use the .bcd file extension when saving the file in your home directory.
6. Click **Save**.
The dialog box closes and the new name is displayed in the *.bcd list. It replaces the initial filename.
7. Select **File** → **Save Changes**, then **File** → **Reset** to display the updated list of target description files.
8. Expand the group (*.bcd) Board/Chip Definitions to show the current list of target descriptions. The list includes the new .bcd file, that is AM.bcd.
9. Delete the contents of the new .bcd file group to avoid problems caused by old or inappropriate settings:
 - a. Expand the new .bcd file group.
 - b. Right-click on the contents, for example, *BOARD=AP, to display the context menu.
 - c. Select **Delete** from the context menu to delete the BOARD group in the file.
Ensure that you delete all groups, or settings, that appear in the new .bcd file group.

10. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

If you copy the file in Windows Explorer, select **Target** → **Connection Properties...** to display the Connection Properties window and then follow this procedure from step 8 onwards.

Note

The general layout and controls of the RealView Debugger settings windows are described in the online help topic *Changing Settings*.

13.3.3 Creating and naming a board, chip, or component

To configure your target, within the *.bcd file that you created in *Creating a new .bcd file* on page 13-15, you must create a BOARD, a CHIP, or a COMPONENT group.

To create a BOARD, a CHIP, or a COMPONENT group in a .bcd file:

1. Right-click on the name of the *.bcd file to display the context menu. For example, right-click on the entry ... \AM.bcd.
2. Select **Make New Group...** from the context menu to display the Group Type/Name selector dialog box.
3. Select the type of group you want to use from BOARD, CHIP, or COMPONENT. For this example, select CHIP.
4. In the Group Name data field change the name from new to something suitable for your target, using only alphanumeric characters, underscore _, and dash -. For this example, enter S5471KT.
5. Click **OK** to create the group. This collapses the AM.bcd entry so that the new group is hidden.
6. Expand the .bcd file group to see the new CHIP=S5471KT group, shown in Figure 13-13 on page 13-18.

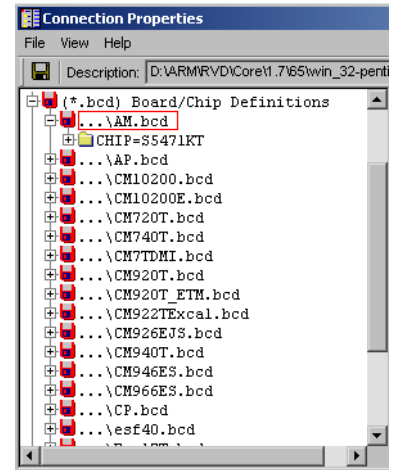


Figure 13-13 Viewing the new group in the .bcd file

7. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Use the method described here to set up as many configuration groups as you require. It is recommended that you do this before creating, or modifying, any CONNECTION groups, as described in *Example descriptions* on page 13-22.

13.3.4 Creating multiple board groups and linking to another board group

To create the structure shown in Figure 13-11 on page 13-12 you must first ensure that the *.bcd files exist:

1. Create a new .bcd file, called ether.bcd as described in *Creating a new .bcd file* on page 13-15.
2. Create a new BOARD group in ether.bcd file, called EtherRouter, as described in *Creating and naming a board, chip, or component* on page 13-17.
3. Create a new .bcd file, called AMDLANCE.bcd as described in *Creating a new .bcd file* on page 13-15.
4. Create a new CHIP group in the AMDLANCE.bcd file, called AMDLANCE, as described in *Creating and naming a board, chip, or component* on page 13-17.

Splitting out the target descriptions in this way makes it easier to reuse configuration details on different connections. Remember to save your changes and reset the board file contents.

To reference these files from your board file:

1. In the Connection Properties window, expand the connection that you are using, for example CONNECTION=testconn2.
2. Right-click on the BoardChip_name entry, in the right pane, to display the **BoardChip** context menu shown in Figure 13-4 on page 13-8.
In this example, the list includes the BOARD called EtherRouter and the CHIP called AMDLANCE.
3. Select the name of the required group, for example **EtherRouter**. A new entry is displayed in the right pane with an asterisk * beside it, shown in Figure 13-14.

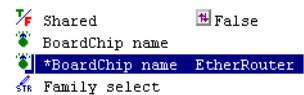


Figure 13-14 Setting up EtherRouter

4. Right-click on the *BoardChip_name EtherRouter entry, in the right pane, and select Jump to Definition from the **BoardChip** context menu.
This expands the group (*.bcd) Board/Chip Definitions to show the current list of target descriptions, which includes the new .bcd files, that is ether.bcd and AMDLANCE.bcd. The ...\ether.bcd group is expanded, and the BOARD=EtherRouter entry is selected.
5. Right-click on the BoardChip_name entry in the right-hand pane to see the **BoardChip** context menu, shown in Figure 13-4 on page 13-8.
6. Select the name of a board group, for example KS32C50100. A new entry is added to the right pane with an asterisk * beside it. For example, *BoardChip_name KS32C50100.
7. Right-click on the BoardChip_name that does not have an asterisk again to see the context menu.
8. Select the name of the other board group, for example AMDLANCE. A new entry is added to the right pane with an asterisk * beside it, shown in Figure 13-15.

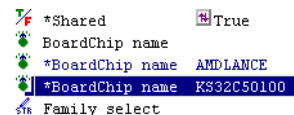


Figure 13-15 Setting up EtherRouter

9. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

When you connect to your RVI/RVI-ME target, the configuration defined in your board groups is applied to the connection. This gives you visibility of all your custom hardware.

Use the method described here to set up as many configuration groups as you require.

13.3.5 Managing configuration settings

You configure your debug target by amending board file entries using the Connection Properties window. This enables you to specify connection behavior, target visibility, image loading parameters, project loading and binding behavior, and disconnect options.

RealView Debugger provides great flexibility in how to configure these settings so that you can control your debug target and any custom hardware that you are using. This means that some settings can be defined in the top-level board file so that they apply to a class of connections, for example CONNECTION=testconn2, or on a per-board (or per-chip) basis using groups in one or more linked Board/Chip definition files, for example CHIP=KS32C50100 in the file Eval7T.bcd.

———— **Note** ————

To avoid conflicts between settings when you reference multiple boards, follow the guidelines given in *Avoiding conflicts between linked board groups* on page 13-6.

To ensure that settings defined in one or more linked .bcd file are used to assemble the target configuration, do not change the default settings contained in the target connection group. For example, if you specify top_of_memory in a linked .bcd file, you must check that the same entry is blank (the default) in the top-level board file:

1. Select **Target** → **Connection Properties...** to display the Connection Properties window.
2. Expand the following entries in turn:
 - a. CONNECTION=testconn2 (*change as required*)
 - b. Advanced_Information
 - c. Default
 - d. ARM_config
3. Ensure that Top_memory is blank.

If the setting contains an entry, right-click to display the context menu, and select **Reset to Empty** to clear the setting.

The interaction between settings in the top-level board file and any linked .bcd files might be important if you are using .bcd files to enable RTOS awareness in RealView Debugger.

———— **Note** —————

The configuration settings `Connect_mode` and `Disconnect_mode` are a special case. See *Specifying connect and disconnect mode* on page 13-29 for details on using these entries.

13.4 Example descriptions

These examples describe how to amend board file entries, using the Connection Properties window, to configure your debug target. They assume you are familiar with the procedures described in the online help topic *Changing Settings*.

In these examples, it is assumed that you are starting with the default board file as installed with the base product. Depending on the type of installation you choose, and your other ARM products, the contents of this file might be different from the one shown here. However, the methods described can be applied to any board file.

In these examples, board file entries are created and renamed. The names used are for illustration only and you can change them as you require. However, it is recommended that you avoid duplicates.

Note

- Do not configure the board file when the debugger is connected to a target.
 - See *Configuration files* on page 12-11 for instructions on making backups of your configuration before you start.
-

The examples are described in the following sections:

- *Setting up an Integrator board and core module* on page 13-23
- *Specifying connect and disconnect mode* on page 13-29
- *Configuring a memory map* on page 13-32
- *Setting up a custom register* on page 13-34
- *Setting up memory blocks* on page 13-39
- *Setting top of memory and stack heap values* on page 13-42
- *Flash programming* on page 13-46.

This section also includes:

- *Restoring your .brd file* on page 13-46 for instructions on restoring your factory settings
- *Troubleshooting* on page 13-46 for instructions on recovering from an incorrectly configured debugger home directory, whether or not you have a backup.

Before you start, remove the linked EtherRouter board description from the connection, if you have been following the instructions in *Creating new target descriptions* on page 13-14. To do this right-click on each *BoardChip_name entry, and select **Delete** from the **BoardChip** context menu. Then select **File** → **Save and Close** to save the changes and close the window.

13.4.1 Setting up an Integrator board and core module

This example demonstrates how to use the Connection Properties window to create a specific Integrator/AP and core module target configuration. It shows how to use a predefined Board/Chip definition file, with extension `.bcd`, to set up your target.

After you set up your target, the example also demonstrates how you can connect to it using RVI/RVI-ME with the Connection Control window, and verify that RealView Debugger can connect to the target.

The example is split into the following sections, which must be executed in this sequence:

1. *Setting up the hardware and debug interface*
2. *Configuring the new target*
3. *Connecting to the new target on page 13-27*
4. *Viewing the new target definition on page 13-27.*

Setting up the hardware and debug interface

The first stage is to set up the hardware and configure the RVI/RVI-ME unit:

1. Ensure that your Integrator/AP and core module are connected and switched on. This example uses the ARM940T processor, but you can use any core module supported by the Integrator/AP.
2. Ensure that you have RVI/RVI-ME installed, and that the RVI/RVI-ME unit is connected and configured for use with RealView Debugger. If you have not configured the RVI/RVI-ME unit, do so now.

Configuring the new target

The next stage is to configure the new target:

1. Start RealView Debugger without connecting to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.
3. Right-click on the `testconn2` entry and select **Connection Properties...** from the context menu to display the Connection Properties window.
4. Right-click on the `CONNECTION=testconn2` entry and select **Make Copy...** from the context menu.

If the testconn2 target vehicle is not visible in the Connection Properties window, you must add it before continuing with this example. See Chapter 14 *Configuring Custom Connections* for full details on how to do this.

5. Use the Group Type/Name selector dialog box, shown in Figure 13-16, to name the new connection. For this example, enter MP3Player.

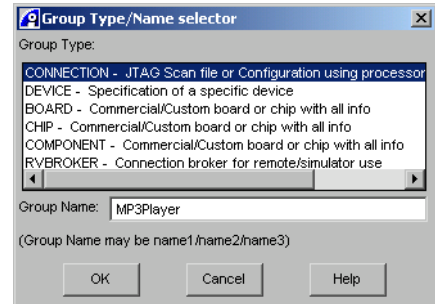


Figure 13-16 Creating a new connection

6. Click **OK**. The new entry, CONNECTION=MP3Player, is added to the List of Entries in the Connection Properties window.
7. Right-click on the Configuration setting for the new connection to display the context menu.
8. Select **Reset to Empty** to delete the current value.
9. Right-click on the Configuration setting again and select **Edit Configuration-File Contents...** the context menu. You are prompted to create a new configuration file either from an existing one, or from an empty one.
10. From the prompt dialog, select **Copy** to copy an existing configuration file. The Select file to copy from dialog box is displayed.
11. Set the Files of type field to **All [*.*)**, and locate the existing rvi.rvc in the directory `install_directory\RVD\Core\...\etc\`.
12. Click **Open**. A Select Name of new file dialog box is displayed. The filename is automatically set to the connection name you specified in step 6. In this example, MP3Player.rvi.
13. Locate your home directory, and click **Save** to save the file. The RVConfig dialog box is displayed.
14. Click **File** → **Save** to save the configuration file.

15. Click **File** → **Exit** to exit the RVConfig dialog box. The location of the MP3Player.rvi file is inserted into the Configuration setting.
16. In the Connection Properties window, select **File** → **Save and Close** to save your changes. The new target is added to the Connection Control window.
17. Right-click on the new MP3Player connection and select **Connection Properties...** from the context menu. This displays the Connection Properties window showing the contents of the new connection, shown in Figure 13-17.

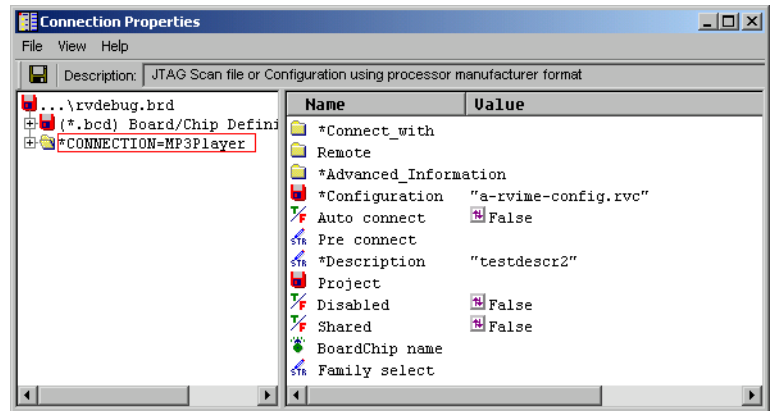


Figure 13-17 Displaying the new MP3Player connection properties

18. Right-click on Description in the right pane to display the context menu. Select **Edit Value** and enter a short description for the new connection, for example Integrator/AP with ARM940 for MP3 product.

———— **Note** —————

Remember to press Enter to complete the entry.

19. Select **File** → **Save and Close** to save your changes. The new target description is added to the Connection Control window.

Linking board groups to the connection

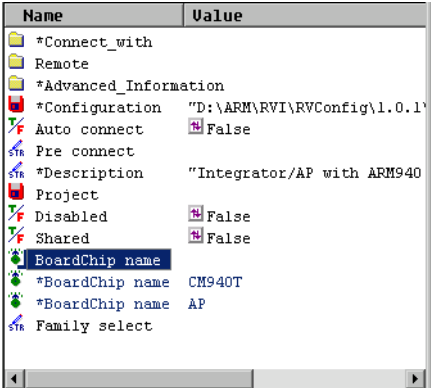
The next stage is to link board groups to the new connection. This is not necessary but gives extended target visibility and enables you to view register contents and manipulate memory.

Note

The connection created in this example is used in other examples in the rest of this chapter. If you do not link the board groups, the contents of the Register pane differ from those shown here.

To link board groups:

1. Right-click on the MP3P1ayer entry and select **Connection Properties...** from the context menu to display the Connection Properties window.
2. Right-click on BoardChip_name in the right pane to display the **BoardChip** context menu.
3. Select **AP** from the context menu to select the Integrator/AP description. A new entry *BoardChip_name AP is added to the pane.
4. Right-click on BoardChip_name (not on *BoardChip_name). The context menu is displayed again.
5. Select **<More...>** to display the List Selection dialog box.
6. Select **CM940T** from the list to select the ARM940T core module description, then click **OK**. The Connection Properties window now shows two BoardChip_name settings, shown in Figure 13-18.



Name	Value
*Connect_with	
Remote	
*Advanced_Information	
*Configuration	"D:\ARM\RV1\RVConfig\1.0.1"
Auto connect	<input type="checkbox"/> False
Pre connect	
*Description	"Integrator/AP with ARM940"
Project	
Disabled	<input type="checkbox"/> False
Shared	<input type="checkbox"/> False
BoardChip name	
*BoardChip name	CM940T
*BoardChip name	AP
Family select	

Figure 13-18 Board groups linked to the new connection

Target configuration settings were copied from the source connection. However, depending on the target hardware, you might have to configure other settings, for example to enable semihosting or to set stack size. See *Configuring a memory map* on page 13-32 for details. If required, you can do this now for the new connection.

7. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Connecting to the new target

The next stage is to connect to the new target board and core module:

1. Select **Target** → **Connect to Target...** to display the Connection Control window.
2. Expand the new MP3Player entry.

The relevant processor name entry is displayed, shown in Figure 13-19.

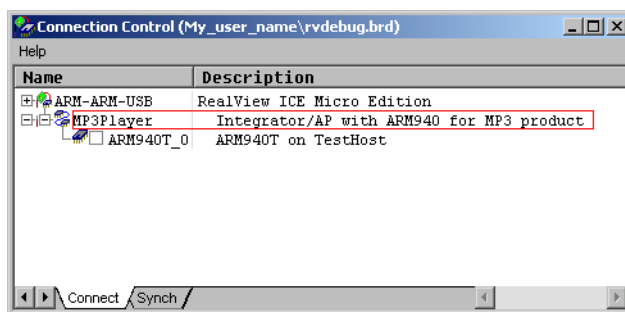


Figure 13-19 Connecting to the new target

3. Click the check box for the processor entry under the new MP3Player entry to connect to the target. RealView Debugger retrieves information specific to the target.

Viewing the new target definition

To view details about the new target hardware:

1. In the Code window, select **View** → **Registers** to display the Register pane. Two new tabs are included at the bottom of the pane, **AP** and **CM940T**.
2. Click on the **AP** tab. RealView Debugger shows the abstraction of the hardware information specific to the Integrator/AP board, shown in Figure 13-20 on page 13-28.

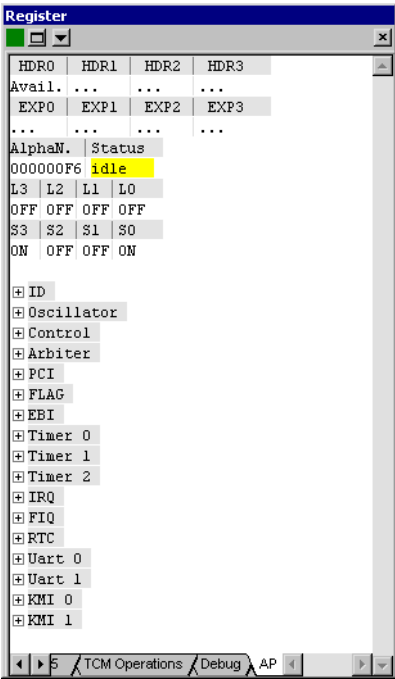


Figure 13-20 AP tab in the Register pane

This tab view enables you to modify your Integrator/AP board features, such as the memory mapped peripherals.

3. To illustrate how RealView Debugger communicates directly with your Integrator/AP board, right-click on the text OFF directly beneath the L2 entry in the Register pane, and select **ON** from the context menu. The relevant LED display on your Integrator/AP board is turned on.
4. Select the **CM940T** tab to see the abstraction of the hardware specific to the core module. The PRESENT status of the Motherboard indicates that the core module is connected to the Integrator/AP board.
For more details on the Register pane, see Chapter 7 *Working with Debug Views*.
5. In the Output pane at the bottom of the Code window, click on the **Cmd** tab. The display includes the line Advanced_info searched...BOARD=AP, BOARD=CM940T indicating that RealView Debugger is using the Integrator/AP and the CM940T board files.

Note

This information is only displayed when you first connect to the target after expanding the access-provider entry (MP3Player in this example).

As a result, the memory map now contains the definitions required to use the Flash memory on the Integrator. See *Flash programming* on page 13-46 for details.

13.4.2 Specifying connect and disconnect mode

If you want to specify how RealView Debugger connects to (or disconnects from) a target processor, you must configure this in your board file. These definitions are contained in the Advanced_Information group for the target processor.

Note

This example uses the connection created in *Setting up an Integrator board and core module* on page 13-23. This example also demonstrates how to manage conflicting settings (see *Managing configuration settings* on page 13-20).

The configuration settings Connect_mode and Disconnect_mode are a special case when used to configure a debug target:

- If a prompt is specified in your board file, or in any .bcd file linked to the connection, it takes priority over any other user-defined setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.
- If a (non-prompt) user-defined setting is specified in your board file and in any .bcd file linked to the connection, the board file setting takes priority.
- A blank entry in the top-level Advanced_Information block ensures that any setting in a linked Board/Chip definition file is used instead. This might be important if you are using .bcd files to enable RTOS awareness in RealView Debugger.

The tasks described in this example illustrate both rules.

Note

The connect (or disconnect) mode that is actually used depends on the target hardware, the target vehicle, and the associated interface software that manages the connection. If you are using RVI/RVI-ME, the unit configuration determines the connect mode and makes the connection. Therefore, the unit configuration might override any settings that you specify in your board file. See *Setting connect mode* on page 2-11 for more details.

To configure connect mode for any hardware debug target:

1. Ensure that RealView Debugger is not connected to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.
3. Right-click on the MP3Player entry and select **Connection Properties...** from the context menu.
4. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)
 - b. Advanced_Information
5. Select the Default entry.
6. Right-click on the Connect_mode entry in the right-hand pane, to display the context menu.

The connection options on the context menu are fixed, and so might include options that are not supported by your target vehicle. If you specify such an option, the debugger prompts you to select an appropriate connection mode when you try to connect. See *Setting connect mode* on page 2-11 for details on how RealView Debugger connects to a target.

7. Select **no_reset_and_no_stop** so that the target processor is not reset when you connect. The running state of the target is unchanged.
8. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

To configure disconnect mode for any hardware debug target:

1. Ensure that RealView Debugger is not connected to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.
3. Right-click on the MP3Player entry and select **Connection Properties...** from the context menu.
4. Expand the following entries in turn:
 - a. (*.bcd) Board/Chip Definitions
 - b. ...\\CM940T.bcd
 - c. BOARD=CM940T
 - d. Advanced_Information

5. Select the ARM940T entry.
6. Right-click on the Disconnect_mode entry in the right-hand pane, to display the context menu.
 The disconnection options on the context menu are fixed, and so might include options that are not supported by your target vehicle. If you specify such an option, the debugger prompts you to select an appropriate disconnection mode when you try to disconnect. See *Setting disconnect mode* on page 2-20 for details on how RealView Debugger disconnects from a target
7. For this example, select **prompt** so that RealView Debugger displays a prompt when you disconnect. This enables you to choose what disconnect mode to use. See *Using a prompt* on page 2-24 for details on using a disconnect prompt in this way.

Note

This setting is stored in the Board/Chip definition file and so is used if you link this file to any other connection.

8. You can override the Disconnect_mode setting by specifying the same setting higher in the configuration hierarchy (see *Managing configuration settings* on page 13-20). To ensure that the setting in the BOARD=CM940T group is used, check for a blank entry. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)
 - b. Advanced_Information
9. Select the Default entry.
10. Ensure that Disconnect_mode is blank.
 If the setting contains an entry, right-click to display the context menu, and select **Reset to Empty** to clear the setting.

Note

If a prompt is specified in your board file, or in any .bcd file linked to the connection, it takes priority over any other user-defined disconnect mode setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.

11. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.
12. To test your changes:
 - a. Connect to your target and load an image, for example dhrystone.axf.

- b. Run your image.
- c. Disconnect from your target to view the Disconnect Mode prompt.
See *Disconnect (Defining Mode)*... on page 2-22 for details on using these options.

———— **Note** —————

This behavior also applies to connect mode, that is, a blank entry in the top-level Advanced_Information block ensures that any setting in a linked Board/Chip definition file is used instead. This might be important if you are using .bcd files to enable RTOS awareness in RealView Debugger.

13.4.3 Configuring a memory map

If you want to set up a memory map that is used automatically when you connect to a target processor, you must configure this in your board file. The memory definition is contained in the Advanced_Information group for the target processor.

———— **Note** —————

This example uses the connection created in *Setting up an Integrator board and core module* on page 13-23. However, in this example, the linked AP and CM940T board groups are removed to ensure a default memory mapping.

To configure a memory map for any hardware debug target:

1. Ensure that RealView Debugger is not connected to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.
3. Right-click on the MP3Player entry and select **Connection Properties...** from the context menu.
4. Right-click on the *BoardChip_name AP setting in the right pane, and select **Delete** from the context menu.
Repeat this for the *BoardChip_name CM940T setting.
5. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)
 - b. Advanced_Information
 - c. Default
 - d. Memory_block

6. Right-click on the Default entry, under Memory_block, to display the context menu.
7. Select **Make Copy...** from the context menu to describe the memory map for the chosen target. Give this entry a suitable name, for example SSRAM, and click **Create**.
8. Click on the new SSRAM entry in the left pane to display it in the right pane.
9. Set the value of Start, in the right pane, to 0x0.
10. Set the value of Length to 0x20000.
11. Set the value of Description to Static RAM as shown in Figure 13-21.

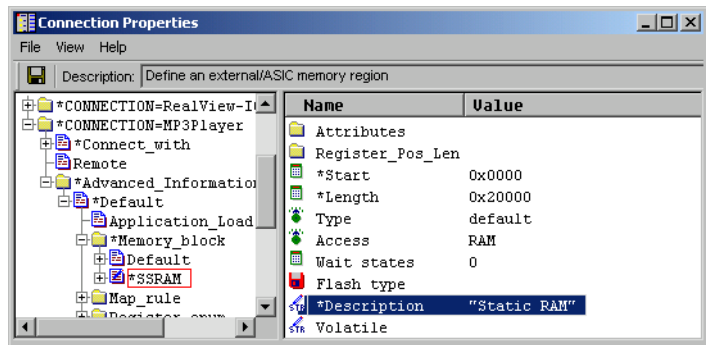


Figure 13-21 Viewing the contents of the new group

12. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.
13. Connect to your target and load an image, for example dhrystone.axf.
14. Select **View** → **Memory Map** tab to view the new memory map, shown in Figure 13-22, before loading an image.

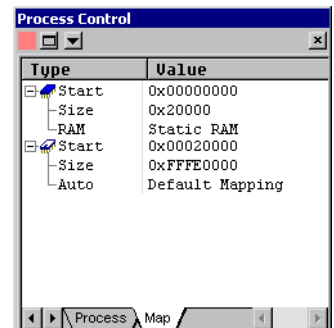


Figure 13-22 New memory map in the Process Control pane

13.4.4 Setting up a custom register

This example describes the steps to follow to specify a register MYREG, that appears as a new tab in the Register pane. It also describes how to set up named bit fields in this register. To set up the custom register, you must make changes in the Memory_block, Register, Register_enum, and Register_Window groups.

———— Note ————

This example uses the connection created in *Setting up an Integrator board and core module* on page 13-23. However, in this example, the AP and CM940T board groups are removed to ensure a default registers view.

If you have set up a memory map as described in *Configuring a memory map* on page 13-32, then delete the SSRAM entry before you begin.

In this example:

- The memory region used for the new register is called REGS and is addressed from 0x10000000-0x107FFFFF.
- The custom register, named MYREG, has an offset of 0x20 from the base of the memory block.
- The custom register, MYREG, has four bit fields. These are used as indicators of the state of the register and are named INDICATORS. They are labeled IND1, IND2, IND3, and IND4.

The example is split into the following sections, which must be executed in this sequence:

1. *Setting up the configuration*
2. *Creating enumerations for the register values* on page 13-36
3. *Creating the register descriptions* on page 13-36
4. *Creating the register tab* on page 13-38
5. *Displaying the register* on page 13-38.

Setting up the configuration

In this stage, you set up a memory group that provides the base address for the new registers:

1. Ensure that RealView Debugger is not connected to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.

3. Right-click on the MP3Player entry and select **Connection Properties...** from the context menu.
4. If the connection has the linked board files from *Setting up an Integrator board and core module* on page 13-23, right-click on the *BoardChip_name AP setting in the right pane, and select **Delete** from the context menu.
Repeat this for the *BoardChip_name CM940T setting.
5. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)
 - b. Advanced_Information
 - c. Default
6. Expand the Memory_block group.
7. If the connection has the SSRAM memory block defined from *Configuring a memory map* on page 13-32, right-click on the SSRAM entry in the left pane, and select **Delete** from the context menu.
8. Rename the Default entry under Memory_block to REGS.
9. Click on the REGS entry, in the left pane, to display the group contents.
10. Set the value of Start, in the right pane, to 0x10000000.
11. Set the value of Length to 0x800000.
12. Set the value of Description to I/O Registers, (shown in Figure 13-23).

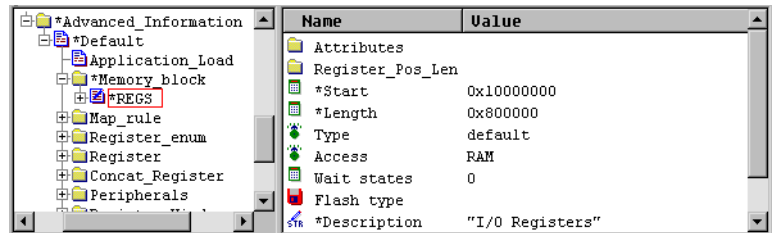


Figure 13-23 Configuring REGS

13. Select **File** → **Save Changes** to save these changes.

Creating enumerations for the register values

In this stage you set up enumerations, or names for specific values, that are used when the register value is displayed:

1. Expand Register_enum in the left pane.
2. Right-click on Register_enum and select **Make New...** to create a new group. Name this E_SWITCH.
3. Click on the E_SWITCH entry, in the left pane, to display the group contents.
4. Set the value of Names, in the right pane, to 0n,0ff.

Note

By default, enumerations are numbered 0,1,2,... However, if you want to give your enumerations specific values, specify them as "enum1=va11,enum2=va12,...". For this example, if you want 0n to equal 7 and 0ff to equal 0 (zero), then enter "0n=7,0ff=0".

5. Rename the Default entry under Register_enum to E_ENABLE.
6. Click on the E_ENABLE entry, in the left pane, to display the group contents.
7. Set the value of the Names entry, in the right pane, to Disable,Enable, shown in Figure 13-24.

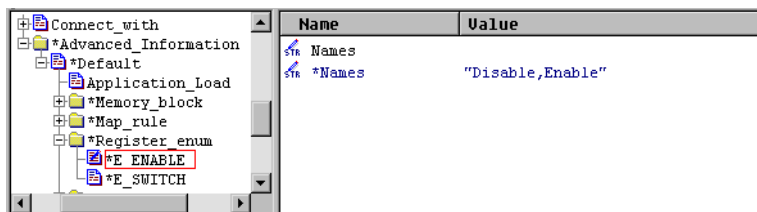


Figure 13-24 Creating enumerations

8. Select **File → Save Changes** to save these changes.

Creating the register descriptions

In this stage you create descriptions of each register:

1. Expand the Register group.
2. Rename the Default entry under Register to Newreg.
3. Select the Newreg group.

4. Set the value of Base to REGS.
5. Set the value of Start to 0x20, that is the offset from the base address of the REGS memory block.
6. Expand the Newreg group.
7. Expand the Bit_fields group. You are now going to set up four bit fields.
8. Rename the Default entry under Bit_fields to IND1.
9. Use **Make Copy...** on IND1. The dialog suggests the name IND2. Click **Create**.
10. Use **Make Copy...** on IND2. The dialog suggests the name IND3. Click **Create**.
11. Use **Make Copy...** on IND3. The dialog suggests the name IND4. Click **Create**.
12. Click IND1, in the left pane and set these values, shown in Figure 13-25:
 - Position=0 (this is the default)
 - Size=1
 - Enum=E_ENABLE.

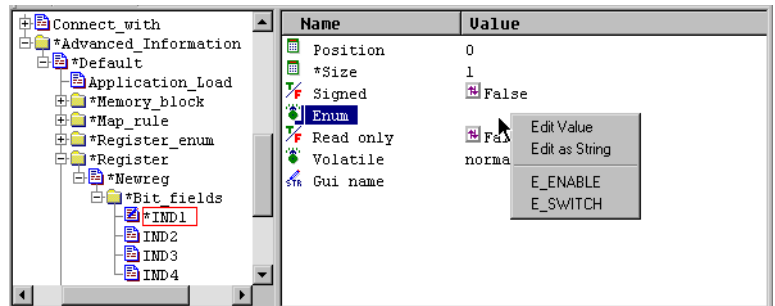


Figure 13-25 Creating bit field descriptions

13. Click on the IND2 entry and set these values:
 - Position=1
 - Size=1
 - Enum=E_SWITCH.
14. Click on the IND3 entry and set these values:
 - Position=2
 - Size=4.
15. Click on the IND4 entry and set these values:
 - Position=6
 - Size=4.

16. Select **File** → **Save Changes** to save the these changes.

Creating the register tab

In this stage you create a Register_Window group to display the new register in the Register pane:

1. Expand the Register_Window group.
2. Rename the Default entry under Register_Window to MYREG. This is the name of the new tab in the Register pane.
3. Click on MYREG, in the left pane.
4. Click on MYREG, in the left pane.
5. Set the Line entry, to Newreg.
6. Use **Make New...** on *Line to create a new *Line entry.
7. Set the *Line entry, to _INDICATORS. Literals entered in *Line (or Line) must be preceded by an underscore.
8. Use **Make New...** on *Line “_INDICATORS” to create a new *Line entry.
9. Set the new *Line to IND1,IND2,IND3,IND4.

The Connection Properties window looks like Figure 13-26.

Name	Value
Line	
*Line	"_INDICATORS"
*Line	"IND1,IND2,IND3,IND4"

Figure 13-26 The MYREG group

All board file entries are now complete.

10. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Displaying the register

In the last stage, display the new register in the Register pane:

1. Connect to your target.
2. Select **View** → **Registers** to view the new tab, **MYREG**, shown in Figure 13-27 on page 13-39.

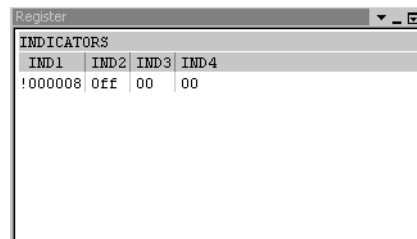


Figure 13-27 MYREG in the Register pane

13.4.5 Setting up memory blocks

This example uses the connection created in *Setting up an Integrator board and core module* on page 13-23. However, the AP and CM940T board groups are removed to ensure a default memory mapping.

This example assumes that you have worked through *Setting up a custom register* on page 13-34, because it uses some of the configuration details set up in that example and saved in the board file.

This example describes how to set up two memory blocks that are activated at different times according to the value of a register. It uses the Newreg register created in *Setting up a custom register* on page 13-34. This is displayed in the **MYREG** tab in the Register pane.

This example also describes setting a memory rule to specify how the memory is used. When IND1 is set to Enabled, MEM2 is activated. Otherwise, MEM1 is used. The example is split into these sections, which must be executed in this sequence:

1. *Defining the memory blocks*
2. *Defining the memory rules* on page 13-41.

Defining the memory blocks

The first stage is to define the two Memory_blocks named MEM1 and MEM2:

1. Select **Target** → **Connect to Target...** to display the Connection Control window.
2. Ensure that RealView Debugger is not connected to a target.
3. Right-click on the MP3Player entry and select **Connection Properties...** from the context menu.
4. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)

- b. Advanced_Information
 - c. Default
 - d. Memory_Block
5. Right-click on the REGS entry, in the left pane, and select **Make New...** from the context menu to see the prompt shown in Figure 13-28.

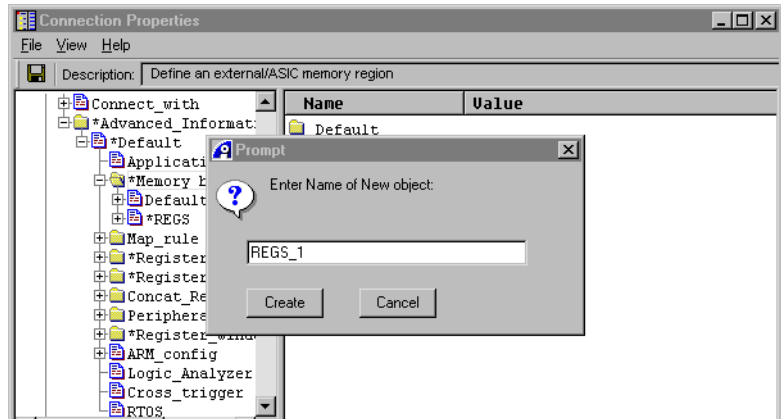


Figure 13-28 Creating a new memory block

6. Enter a new name for this entry, for example MEM1, and click **Create**.
7. Click on the MEM1 entry, in the left pane.
8. Set the value of Start to 0x0 (this is the default).
9. Set the value of Length to 0x80000.
10. Set the value of Description to Fast Static RAM.
11. Set the value of Access entry to RAM (this is the default).
12. Use **Make Copy...** on MEM1 to create a new group, MEM2.
13. Click on the MEM2 entry, in the left pane, to display the settings values.
14. Set the value of Access to ROM.
15. Set the value of *Description to Slow Boot ROM.
16. Select **File** → **Save Changes** to save these changes.

Defining the memory rules

The second stage is to define the rules that control which memory block is used:

1. Expand the Map_rule group.
The map rule defines which memory block to use. In this example, MEM2 is activated if IND1 is set to Enabled (one). Otherwise MEM1 is used.
2. Click on the Default entry.
3. Set the following values, shown in Figure 13-29:
 - Register to Newreg (use the context menu).
 - Mask to 0x0001 (to check IND1 value only)
 - Value to 0 (IND1 set to Disabled)
 - On_equal to MEM1.

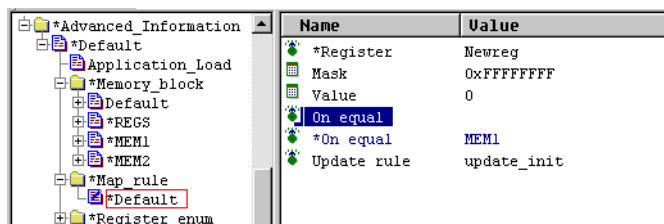


Figure 13-29 Creating a map rule

4. Rename the Default entry of Map_rule to RULE1.
5. Use **Make Copy...** on RULE1, to create a new group, RULE2.
6. Click on RULE2 and set the value of Value to 1 (IND1 set to Enabled).
7. Set the value of *On_equal to MEM2, shown in Figure 13-30.

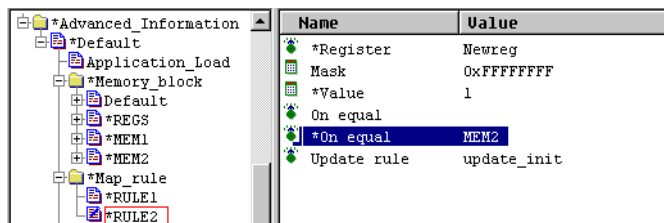


Figure 13-30 Settings for the second map rule

All board file entries are now complete.

8. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.
9. Connect to your target.
10. In the Code window, select **View** → **Registers** to view the **MYREG** tab.
11. Select **View** → **Memory Map** tab to view the **Map** tab, shown in Figure 13-31.

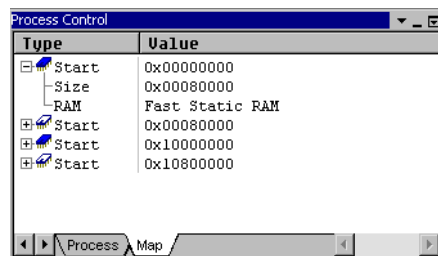


Figure 13-31 New memory block in the Map tab

12. Set the IND1 value to Enable to activate the memory rule (RULE2) and so change the memory map. The Start entry in the **Map** tab changes to a ROM entry, indicated by a yellow icon.

13.4.6 Setting top of memory and stack heap values

This example demonstrates how you can set permanent `top_of_memory` and stack heap values for a given target using your board file. It shows how to do this by updating the information in the `ARM_config` group of the `Advanced_Information` block for the connection. After you have defined the settings, they are used whenever you connect to the target with RealView Debugger.

Setting top of memory

You can set the value of `Top_memory` in a `BOARD` group used to configure the target or in the `CONNECTION` entry you use to connect to the target. The method described here applies to both. This example uses an Integrator/AP board with an ARM940T core module, but the procedure for amending the settings is the same for any target.

———— Note ————

This example uses the connection created in *Setting up an Integrator board and core module* on page 13-23. However, the AP, AM, and CM940T board groups are not linked to ensure a default memory mapping. If you do link the board groups, the contents of the

Register pane differ from those shown here because the value of `top_of_memory` is defined in the linked CM940T board group. This value overrides any value that you define in the CONNECTION entry.

The `Top_memory` variable is used to enable the semihosting mechanism to return the stack base address. You can create your own settings to specify the bottom of the stack address, the size of the stack, the bottom of the heap address, and the size of the heap. If you do not set these values manually, RealView Debugger uses the target-dependent defaults. If your application is scatterloaded, you must define the stack and heap limits in `__user_initial_stack_heap`.

Note

The value of `Top_memory` must be higher than the sum of the program base address, program code size, and program data size. If set incorrectly, the program might crash because of stack corruption or because the program overwrites its own code.

There is no requirement that the address specified by `top_of_memory` is at the true top of memory. A C or assembler program can use memory at higher addresses.

To set the `top_of_memory` in the CONNECTION:

1. Ensure that RealView Debugger is not connected to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.
3. Right-click on the MP3Player entry and select **Connection Properties...** from the context menu.
4. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)
 - b. Advanced_Information
 - c. Default
5. Click ARM_config in the left pane.
6. Set the value of `Top_memory`, in the right pane, as required. For example, set it to `0x40000` if your target has 256KB of RAM starting at location 0.

Note

Be sure to specify a value that is supported by your debug target.

When you load a program, the debugger sanity-checks Top_memory by checking that the words below Top_memory are writable. It issues a warning if they are not. However, your program might require much more RAM than the debugger checks for.

7. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Whenever you load a program compiled with the standard ARM C library to this target, the top_of_memory value you have set is used.

The default value of Top_memory for ARM processors is 0x20000. For details on how this variable is relevant to ARM targets, see the internal variable descriptions sections in the user guide that accompanies your interface.

Setting stack heap values

Table 13-1 gives a summary of the stack heap settings.

Table 13-1 Stack heap settings

Setting	Description
Stack_Heap.Stack_bottom	Specifies the lowest address of the stack.
Stack_Heap.Stack_size	Specifies the stack size in bytes.
Stack_Heap.Heap_base	Specifies the lowest address of the heap.
Stack_Heap.Heap_size	Specifies the heap size in bytes. This is appropriate only if you are using a two region model.

The stack-related settings are appropriate only if you are using software stack checking.

To set the stack heap values in the CONNECTION:

1. Ensure that RealView Debugger is not connected to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window.
3. Right-click on the MP3P1ayer entry and select **Connection Properties...** from the context menu.

4. Expand the following entries in turn:
 - a. CONNECTION=MP3Player (*change as required*)
 - b. Advanced_Information
 - c. Default
 - d. ARM_config
5. Click on the Stack_Heap group in the left pane to display the contents, shown in Figure 13-32.

Name	Value
Stack bottom	<above heap>
Stack size	0x2000
Heap base	
Heap size	0x4000

Figure 13-32 Settings in the Stack_Heap group

This shows the currently selected size and location of the stack and heap. A blank or zero Heap_base value is modified by the ARM C library runtime code, setting it to the address of the end of program data space, shown in Figure 13-33.

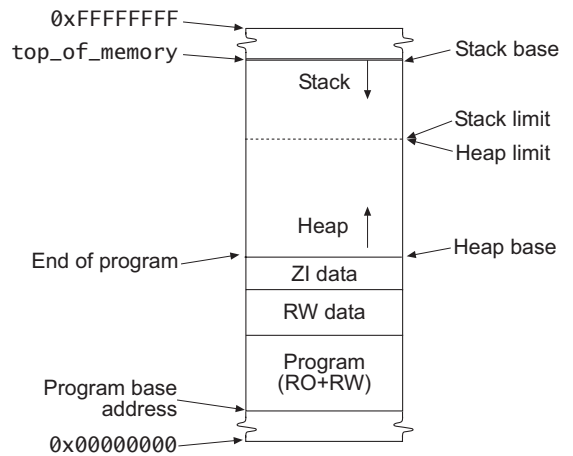


Figure 13-33 Relating top_of_memory to single section program layout

6. If you require more control over the stack and heap location for a semihosted program, set the values as required.

If your application requires control over the stack and heap location, or if the application is scatterloaded, the application must include a user-defined function, `__user_initial_stackheap`, that defines the stack and heap limits.

7. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Whenever you load a program compiled with the standard ARM C library to this target, the `top_of_memory`, `stack`, and `heap` values you have set are used.

The default value of `Top_memory` for ARM processors is `0x20000`. For details on how this variable is relevant to ARM targets, see the internal variable descriptions sections in the user guide that accompanies your interface.

13.4.7 Flash programming

Programming the Flash on the Integrator/AP involves:

- assigning the BCD file that describes the memory map and specifies a *Flash Method* (FME) file to the target connection.
- programming the image to the Flash device.

Chapter 16 *Programming Flash with RealView Debugger* gives detailed information and instructions on how to program Flash if you have:

- an ARM board that is supported by RealView Debugger
- a custom board that uses one of the Flash types supported by RealView Debugger
- a custom board that uses a Flash type not supported by RealView Debugger.

13.4.8 Restoring your .brd file

If you have completed these examples and you want to return to the factory settings:

1. Exit RealView Debugger.
2. Delete your RealView Debugger home directory.

When you restart RealView Debugger it creates a new default configuration for you.

13.4.9 Troubleshooting

If your working versions of configuration files are accidentally erased, or become corrupted, RealView Debugger might be unable to use them. In this case, making a connection to your chosen target is not possible.

You can do one the following:

- If you have made a backup of your configuration, restore it as described in *Saving and restoring connection properties* on page 12-14.

- If it is acceptable to lose all of the configuration settings, program preferences, workspaces and other information that is stored in the debugger home directory, you can delete it:
 1. Exit RealView Debugger.
 2. Locate the home directory the debugger is using.
See *The home directory* on page 12-13 for more details.
 3. Use Windows Explorer to rename or delete the home directory.
You might want to move or rename it before deleting so that if you make a mistake you can recover selected files.
 4. Restart RealView Debugger. It creates a new, default copy of the debugger home directory as it starts up.
- If there are configuration items that you want to try to keep:
 1. Exit RealView Debugger.
 2. Using Windows Explorer, display the home directory the debugger is using.
See *The home directory* on page 12-13 for more details.
 3. Using a second Windows Explorer window, locate the RealView Debugger installation directory.
See *The install directory* on page 12-12 for more details.
 4. Use the hints given in *Using manual file or directory backups* on page 12-15 to copy files from the default settings directory \etc to your debugger home directory. Some of the *.cnf files have no default in \etc, and are recreated as required. If you believe it is causing problems, delete the version in your home directory and let the debugger recreate it when you next connect.
 5. Restart RealView Debugger.

Chapter 14

Configuring Custom Connections

This chapter describes how you can configure the connection that RealView® Debugger makes to your target. It includes information on the board file groups CONNECTION and DEVICE, and explains how connections to different targets are configured. It contains the following sections:

- *Working with connection properties* on page 14-2
- *Working with RVI/RVI-ME targets* on page 14-7

14.1 Working with connection properties

As described in *About target connections and configuration* on page 12-2, you configure the way that RealView Debugger connects to, and interacts with, your debug target using connection properties contained in board file entries. A debug target might be a simulator, an emulator, or a development board installed on your host workstation. Using connection entries enables you to configure:

- debugger to target connection details, such as interface type and instance, TAP controller positions, and connection interface address
- debugger actions taken when a connection is made, for example running commands and opening projects.

This section describes how to work with connection entries. It contains the following sections:

- *Connection entries in the Connection Control window* on page 14-3
- *Enabling or disabling a board file entry* on page 14-4
- *Changing entries containing user-information values* on page 14-5
- *Restoring board file entry defaults* on page 14-5.

In the examples in the rest of this chapter, you are changing your board file. This is stored in your RealView Debugger home directory. Target configuration files might also be stored in this directory, for example .rvc files or .cnf files.

It is recommended that you back up this directory before starting the examples described in this chapter, so that you can restore your original configuration later. For details see:

- *Configuration files* on page 12-11 for instructions on making backups of your configuration
- *Restoring your .brd file* on page 13-46 for instructions on restoring a default configuration
- *Troubleshooting* on page 13-46 for instructions on recovering from an incorrectly configured debugger home directory, whether or not you have a backup.

Note

When you are following these examples, do not configure the board file when the debugger is connected to a target.

There are descriptions of the general layout and controls of the RealView Debugger settings windows, including the Connection Properties window, in the RealView Debugger online help topic *Changing Settings*. This chapter assumes you are familiar with the procedures described in this help topic.

14.1.1 Connection entries in the Connection Control window

Board file entries that are enabled form the basis of the information displayed in the Connection Control window, shown in Figure 14-1.

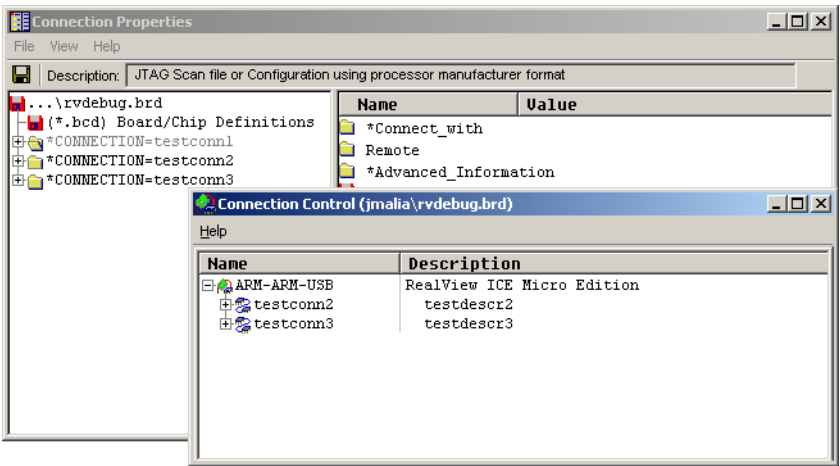


Figure 14-1 Connection properties entries in the Connection Control window

In Figure 14-1 the entry CONNECTION=testconn1 has been disabled in the board file. Therefore, this entry does not appear in the Connection Control window. Other connection entries are enabled and so are visible, for example CONNECTION=testconn2.

You can enable and disable any connection entry in the board file if required. When you view the Connection Properties window, disabled entries are grayed out in the left pane, the List of Entries pane. Disabled entries can be edited in the same way as enabled entries and then enabled when available for connection. See *Enabling or disabling a board file entry* on page 14-4 for details on how to do this.

If you make changes to values in the Connection Properties window, an asterisk is added to each entry, in the left or the right pane, to show that the defaults have changed. You can restore the default settings and so cancel any changes. See *Restoring board file entry defaults* on page 14-5 for details on how to do this.

When you display the Connection Properties window, the left pane shows the top-level entries specifying the supported vehicles, for example Board/Chip Definitions. Board file entries might have duplicate names because entries are uniquely identified through the combination of three elements:

- the CONNECTION entry
- the name of the manufacturer of the simulator, emulator, or board
- the host workstation I/O device address, IP address (for remote connections), or ID, of the emulator or board.

For example, assume that you have a target board named MP3Player that you want to use with two different emulators. The board file entry name for each is MP3Player to reflect the target. However, the entries are differentiated by the type of connection (emulator type), and I/O device connection addresses.

———— **Note** ————

You can create your own custom entries in this hierarchy using other types of entry, for example BOARD, CHIP, COMPONENT, or DEVICE. However, if you are creating custom, lower-level entries, it is recommended that you avoid duplicate names.

For full information on the contents and values contained in different types of board file entries, both default and custom, see Appendix B *Configuration Properties Reference*.

14.1.2 Enabling or disabling a board file entry

To disable a board file entry so that the target it represents is no longer offered for selection in the Connection Control window:

1. Start RealView Debugger without connecting to a target.
2. Select **Target** → **Connection Properties...** to display the Connection Properties window.
Enabled entries in the left pane are displayed in regular type, and those that are disabled are grayed out.
3. Expand the connection that you are using, for example RVBROKER=localhost.
It becomes the selected entry and its contents are displayed in the right pane.
4. Right-click on the Disabled entry, in the right pane, and select **True** from the menu.
5. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

6. Select **Target → Connect to Target....** to display the Connection Control window where this entry is no longer available.

14.1.3 Changing entries containing user-information values

Where entries contain user-information values, you can customize them as follows:

1. Select **Target → Connection Properties...** to display the Connection Properties window.
2. Expand the entry, in the left pane.
3. Click on the CONNECTION=testconn1 entry, in the left pane. It becomes the selected entry and its contents are displayed in the right pane.
4. Right-click on the Description entry, in the right pane, and select **Edit Value...** from the context menu.

The text defined by this entry appears in the Description in the Connection Control window, shown in Figure 14-1 on page 14-3. Enter a new description, for example, mytestconnection, and press Enter.

5. Select **File → Close Window** to close the Connection Properties window without saving this change. This generates a dialog box warning that contents have changed and giving you the option of saving them. Do not save this change.

This example demonstrates:

- the Description setting has been changed in your user-specific copy of the board file (... \home\My_user_name\rvdebug.brd), and the default board file in ... \etc\rvdebug.brd is not changed
- because the setting has been changed, an asterisk marks the entry
- RealView Debugger warns you if you have not saved changes you have made to the board file settings.

14.1.4 Restoring board file entry defaults

An entry starts with an asterisk when you have changed that entry from the default. For group entries, this might mean that a value lower down in the hierarchy has been changed.

To restore the default values for an entry:

1. Select **Target → Connection Properties...** to display the Connection Properties window.

2. Locate the entry that you want to reset to the default value.

For example, in *Enabling or disabling a board file entry* on page 14-4, the RVBROKER=localhost entry was disabled. This means that it now contains a custom setting, that is the default setting has been changed. Click on the RVBROKER=localhost entry, in the left pane. It becomes the selected entry and its contents are displayed in the right pane.

3. Right-click on the entry in the right pane, and select the appropriate option from the context menu:
 - **Reset to Default** for entries that have a default value in the default board file. This sets the value of the setting to that defined in the default board file.
 - **Reset to Empty** for entries that have no value by default. This clears the value for the entry.

The asterisk is removed.

For the example in step 2, select **Reset to Default** for the *Disabled entry. This sets the value of this setting to False as defined in the default board file.

4. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

The original contents of the Connection Control window are restored.

14.2 Working with RVI/RVI-ME targets

On starting the debugger, the appropriate RVI or RVI-ME connection appears in the Connection Control window. This updates the default board file so that, for RVI-ME, an ARM-ARM-USB target vehicle appears in the Connection Control window to provide the interface to RVI-ME targets, as shown in Figure 14-2:

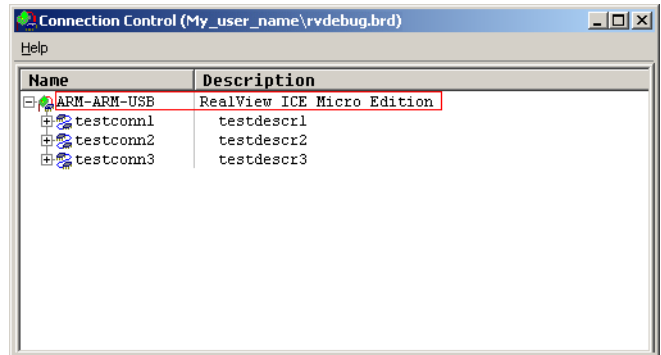


Figure 14-2 RVI-ME targets in the Connection Control window

For RealView ICE, an ARM-ARM-NW target vehicle appears in the Connection Control window.

Note

If the RVI/RVI-ME target vehicle is not visible in the Connection Properties window, you must add it before continuing with this section.

For details on how to configure RealView ICE connections, you should refer to the chapter on configuring a RealView ICE connection in *RealView ICE and RealView Trace Version 1.5 User Guide*.

If you expand the ARM-ARM-USB vehicle, you can see the access-provider connection defined automatically, that is testconn2 shown in Figure 14-2. You can then configure this connection by specifying an appropriate .rvc file, ready to make a connection. However, it is recommended that you use this default connection as the basis for new connections that you create.

This section describes:

- *Creating a new RVI/RVI-ME connection* on page 14-8
- *Configuring an RVI/RVI-ME interface unit* on page 14-10.

14.2.1 Creating a new RVI/RVI-ME connection

This example defines a new RVI-ME connection. You can do this by creating a new connection entry or by copying an existing entry. A similar procedure may be used for RVI connections.

———— Note ————

If you copy a connection entry, certain configuration details are not updated until you close the Connection Properties window. This might mean that some asterisks are not immediately visible in the copy.

This example creates a new connection entry. It assumes that a correctly configured .rvc file exists for the new target and this has been saved in the default RVI-ME installation directory. If you do not have this file, you can follow the example but you must also follow the instructions in *Configuring an RVI/RVI-ME interface unit* on page 14-10 before you can connect to the new target.

To define the new connection:

1. Select **Target** → **Connection Properties...** to display the Connection Properties window.
2. Right-click on the CONNECTION=RVI-ME entry, in the left pane.
3. Select **Make New...** from the context menu.
4. This displays the Group Type/Name selector dialog box, shown in Figure 14-3.

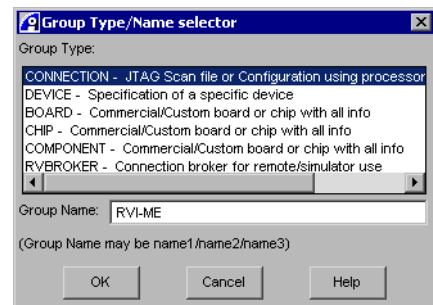


Figure 14-3 Specifying a new CONNECTION group

Leave the type of the new entry unchanged as CONNECTION.

In the Group Name data field change the name from RVI-ME to something suitable for your target, using only alphanumeric characters, underscore _, and dash -, for example RVI-ME_2.

This can be a descriptive name or the name of the new .rvc file that you are going to select, but without the .rvc extension.

5. Click **OK** to confirm your settings and to close the Group Type/Name selector dialog box.

The new entry appears in the left pane of the Connection Properties window. It is automatically selected, and its details are displayed in the right pane. These details are the default for a new CONNECTION and you must change at least the Connect_with/Manufacturer, the Configuration filename and target Description. The next steps explain how to make these changes.

6. In the right pane of the Connection Properties window, right-click on the Configuration entry and select **Edit as Filename** from the context menu.

The Enter New Filename dialog box is displayed to enable you to locate the required .rvc file, for example RVI-ME_2.rvc.

7. Click **Save** to confirm your entries and to close the Enter New Filename dialog box.

The new pathname is displayed in the right pane.

8. In the right pane of the Connection Properties window, right-click on the Description field and select **Edit Value** from the context menu.

Type RVI-ME connection to test hardware board in the entry area and press Enter.

This is the description displayed in the Connection Control window and Connection Properties window to identify the new target.

9. In the right pane of the Connection Properties window, right-click on the Connect_with entry and select **Explore** from the context menu.

10. In the right pane of the Connection Properties window, right-click on the Manufacturer entry and select the required connection type from the context menu, that is **ARM-ARM-USB**.

If you do not specify this setting, the new connection appears in the Connection Control window but, when you try to connect, RealView Debugger prompts for the connection type.

11. If you want to configure other settings for your target, for example reference a specific .bcd file, make these changes now.

12. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Your new RVI-ME target is now displayed in the Connection Control window.

14.2.2 Configuring an RVI/RVI-ME interface unit

As an example, use RVConfig to configure a new RVI-ME interface unit as described below. A similar procedure may be used for an RVI interface unit.

1. Right-click on your new RVI-ME connection to display the context menu.
2. Select the option **Configure Device Info...** to display the RVConfig dialog box shown in Figure 14-4.

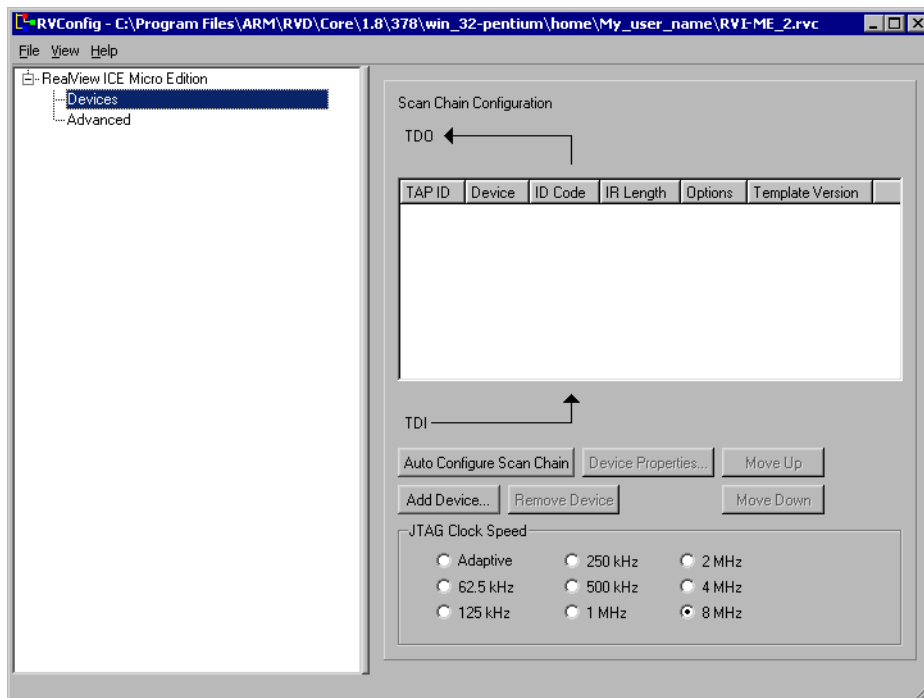


Figure 14-4 Configuring an RVI-ME interface unit

3. Select the device, for example ARM940T, in the left pane, to configure it using the supplied template. For example, you might want to set the Code Sequence Code Address or Code Sequence Code Size, to specify target memory available to RVI-ME.
4. Select Advanced, in the left pane, to configure advanced settings, for example, you might want to click the **Reset On Disconnect (Default)** check box so that the target hardware is reset when you disconnect.

If you are using an Integrator™ board, as in this example, it is recommended that you use the settings shown in Figure 14-5 on page 14-11.

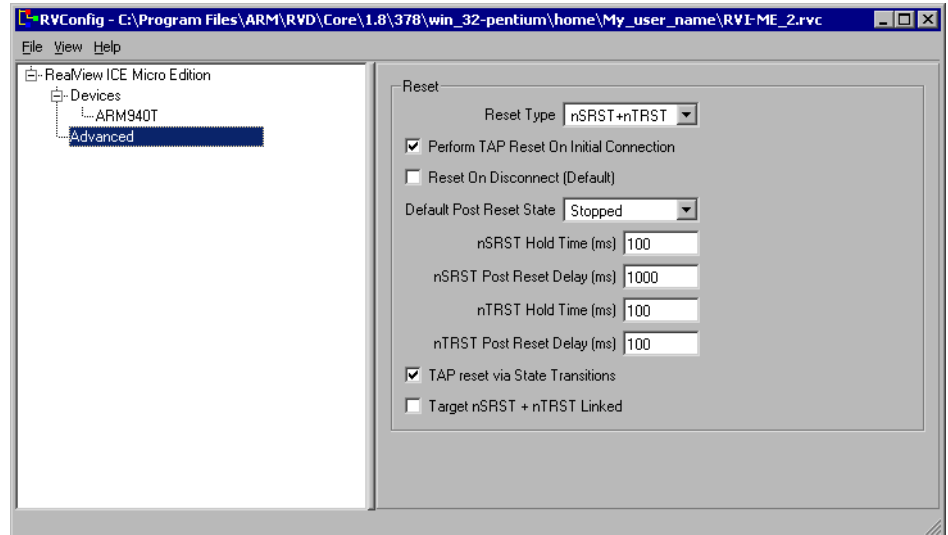


Figure 14-5 Recommended settings for an ARM Integrator board

5. Select **File** → **Save** to save your changes.
 6. Select **File** → **Exit** to close the RVConfig dialog box.
- You can now connect to your new target in the usual way.

Configuring RVI/RVI-ME units for different targets

You might have to use different configuration settings from those shown in this example, depending on your debugging environment. Remember the following when specifying these settings:

- Autoconfiguration does have side effects and might be intrusive. Where this is not acceptable, you must configure the target manually.
- You must use different settings if your application uses interrupts to set up pre-initialization code on the target, for example if you are using *Running System Debug* (RSD).
- The RVI/RVI-ME scan chain configuration lists devices in ascending order of TAP ID.

For full details on using RVConfig, and on how to add new devices to the scan chain, see the chapter that describes configuring a RealView ICE connection in *RealView ICE and RealView Trace User Guide*, or if appropriate, the chapter that describes configuring an RVI-ME connection in *RealView ICE Micro Edition v1.1 User Guide*.

Chapter 15

RTOS Support

This chapter describes the *Real Time Operating System* (RTOS) support available in RealView® Debugger. It contains the following sections:

- *About Real Time Operating Systems* on page 15-2
- *Using RealView Debugger RTOS extensions* on page 15-7
- *Connecting to the target and loading an image* on page 15-22
- *Associating threads with views* on page 15-27
- *Working with OS-aware images in the Process Control pane* on page 15-34
- *Using the Resource Viewer window* on page 15-41
- *Debugging your RTOS application* on page 15-47
- *Using CLI commands* on page 15-58.

Note

RTOS support is available only on Windows platforms.

15.1 About Real Time Operating Systems

Real Time Operating Systems (RTOSs) manage software on a debug target. They are designed for applications that interact with real-world activities where the treatment of time is critical to successful operation. A real-time multitasking application is a system where several time-critical tasks must be completed, for example an application in control of a car engine. In this case, it is vital that the electronic ignition and engine timing are synchronized correctly.

Real-time applications vary in required timing accuracy from seconds to microseconds, but they must guarantee to operate within the time constraints that are set.

Real-time applications can be:

Hard real-time	Failure to meet an event deadline is catastrophic, typically causing loss of life or property. An example is a car engine controller.
Soft real-time	Failure to meet a deadline is unfortunate but does not endanger life or property. An example is a washing machine controller.

In supporting real-world computer systems, an RTOS and the applications using it are designed with many principles in mind, for example:

- The algorithms used must guarantee execution in tightly bounded (but not necessarily the fastest possible) time.
- The applications must guarantee that they do not fail during execution. This in turn implies the RTOS itself does not fail.
- RTOSs supporting hard real-time systems must enable sufficient control over process scheduling to specify and meet the deadlines imposed by the overall system.

An RTOS often uses separate software components to model and control the hardware with which it interacts. For example, a car engine controller might have two components to:

- model the motion of the cylinder, enabling it to control ignition and valve timing
- monitor fuel consumption and car speed and display trip distance and fuel economy on the dashboard.

Using components like this enables the RTOS to schedule jobs in the correct order to meet the specified deadlines.

RTOS jobs can be:

- Processes** Created by the operating system, these contain information about program resources and execution state, for example program instructions, stack, and heap. Processes communicate using shared memory or tools such as queues, semaphores, or pipes.
- Threads** Running independently, perhaps as part of a process, these share resources but can be scheduled as jobs by the RTOS.
A thread can be controlled separately from a process because it maintains its own stack pointer, registers, and thread-specific data.

In single-processor debugging mode, an RTOS might control one or more processes running on a single processor. Similarly, a process can have multiple threads, all sharing resources and all executing within the same address space. Because threads share resources, changes made by one thread to shared system resources are visible to all the other threads in the system.

15.1.1 Debugging an RTOS application with RealView Debugger

Debugging real-time systems presents a range of problems. This is especially true where the software being debugged interacts with physical hardware, because you normally cannot stop the hardware at the same time as the software. In some real-time systems, for example disk controllers, it might be impossible to stop the hardware.

————— Note —————

RealView Debugger can support a single RTOS connection or it can be used to debug multithreaded applications running on multiple processors.

Running and Halted System Debug

RealView Debugger supports different debugging modes, depending on the RTOS you are using:

Halted System Debug

Halted System Debug (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. This debugging mode places no demands on the application running on the target.

However, HSD mode might not be suitable for real-time systems where stopping the debug target might damage your hardware, for example disk controllers.

Running System Debug

Running System Debug (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a *Debug Agent* (DA) that resides on the target and is typically considered to be part of the RTOS. The Debug Agent is scheduled along with other tasks in the system. See *Debug Agent* for details.

RSD mode is intrusive because it uses resources on your debug target and makes demands on the application you are debugging. However, this debugging mode provides extra functionality not available when using HSD, for example, RSD enables you to debug threads individually or in groups, where supported by your RTOS.

RealView Debugger enables you to switch seamlessly between RSD and HSD mode using GUI controls or CLI commands. For details see:

- *Working with OS-aware images in the Process Control pane* on page 15-34
- *Using the Resource Viewer window* on page 15-41
- *Using CLI commands* on page 15-58.

Debug Agent

RSD requires the presence of a Debug Agent on the target to handle requests from the RealView Debugger host components. The Debug Agent is necessary so that the actions required by the host can coexist with the overall functioning of the target RTOS and the application environment. This relationship is shown in Figure 15-1 on page 15-5.

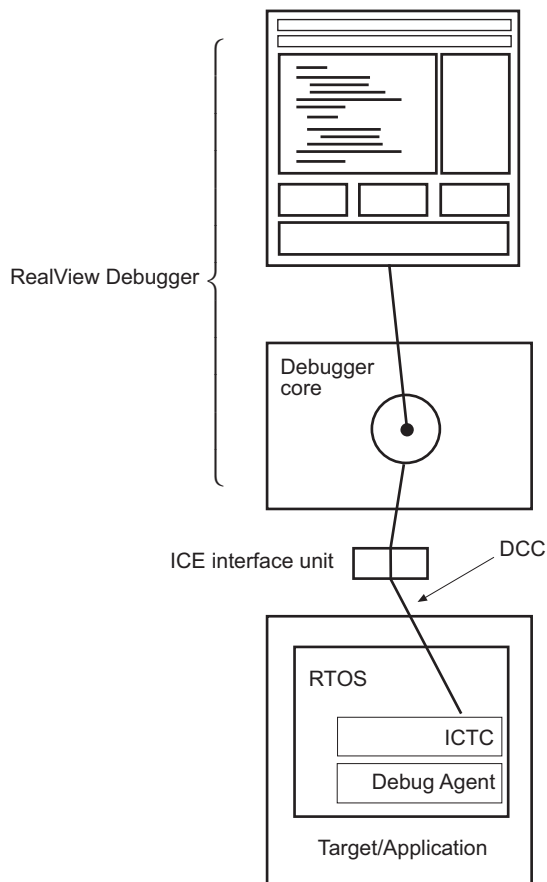


Figure 15-1 RealView Debugger and RTOS components

The Debug Agent and RealView Debugger communicate with each other using the *debug communications channel* (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state. The Debug Agent provides debug services for RealView Debugger and interacts with the RTOS and the application that is being debugged.

Note

A DCC device driver, the *IMP Comms Target Controller* (ICTC), is required to handle the communications between the Debug Agent and RealView Debugger. Depending on the RTOS, the ICTC is part of the Debug Agent or the RTOS.

By interacting with the RTOS running on the target, the Debug Agent can gather information about the system and make modifications when requested by the user, for example to suspend a specified thread.

In summary, the Debug Agent:

- provides a direct communications channel between the RTOS and RealView Debugger, using the ICTC
- manages the list of threads on the system
- enables thread execution control
- manages RTOS objects such as semaphores, timers, and queues
- accesses RTOS data structures during RSD mode.

15.2 Using RealView Debugger RTOS extensions

This section describes how to use RealView Debugger RTOS extensions and configure an RTOS-enabled connection.

Note

RTOS-specific files are not installed with RealView Debugger. RTOS awareness is achieved by using plugins supplied by your RTOS vendor. This means that you must download the files you require after you have installed RealView Debugger. Select **Help** → **ARM on the Web** → **Goto RTOS Awareness Downloads** from the Code window menu for more information.

This section describes:

- *Enabling RTOS support*
- *Creating a new RTOS-enabled connection on page 15-8*
- *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file on page 15-12*
- *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file on page 15-14*
- *Managing configuration settings on page 15-21.*

15.2.1 Enabling RTOS support

Your RTOS vendor supplies plugins to enable RTOS awareness in RealView Debugger:

- a DLL, *.dll
- one or more Board/Chip definition files, *.bcd.

To get started, install the plugins in your root installation:

1. Copy the *.dll file into the *install_directory\RVD\Core\...\lib* directory.
2. Copy the *.bcd file(s) into the *install_directory\RVD\Core\...\etc* directory.

The Board/Chip definition file contains the information required to enable RTOS support in the debugger.

15.2.2 Creating a new RTOS-enabled connection

It is recommended that you create a new connection in the board file to specify your RTOS-enabled target. Although this is not necessary, it means that it is easy to identify the RTOS target and maintains other custom targets that you might configure. This section describes how to set up the new connection.

This example defines a new RVI-ME connection. You can do this by creating a new connection entry or by copying an existing entry. Here, you create a new connection entry.

The example assumes that a correctly configured .rvc file exists for the new target and this has been saved in the default RVI-ME installation directory. If you do not have this file, you can follow the example. However, before you can connect to the new target, you must also follow the instructions describing how to configure an RVI-ME interface unit in Chapter 14 *Configuring Custom Connections*.

To set up the new connection:

1. Start RealView Debugger but do not connect to a target.
2. Select **Target** → **Connect to Target...** to display the Connection Control window, shown in Figure 2-1 on page 2-2.



You can also click the **Connection Control** button in the Connect toolbar to display the Connection Control window quickly. If the window is hidden, click the button twice.

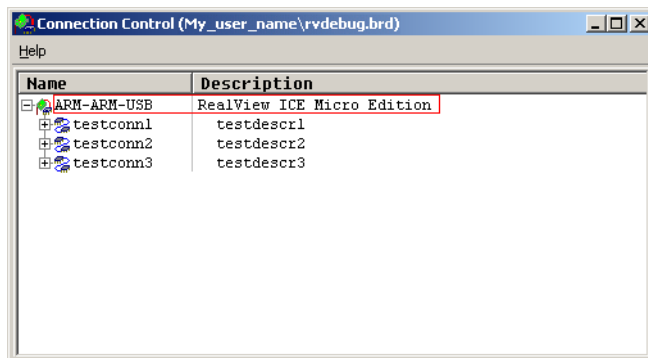


Figure 15-2 Connection Control window

Figure 2-1 on page 2-2 shows the default connections set up after you first install RealView Debugger. The contents of this window depend on the autodetected targets available to you. If you have installed a Custom configuration your window looks different.

Note

If the RVI-ME target vehicle is not visible in the Connection Properties window, you must add it before continuing with this section. See Chapter 14 *Configuring Custom Connections* for full details on how to do this.

3. Right-click on the connection that you want to use. This example uses RVI-ME to access the ARM® JTAG debug tool.
4. Select **Connection Properties...** from the context menu to display the Connection Properties window, shown in Figure 15-3.

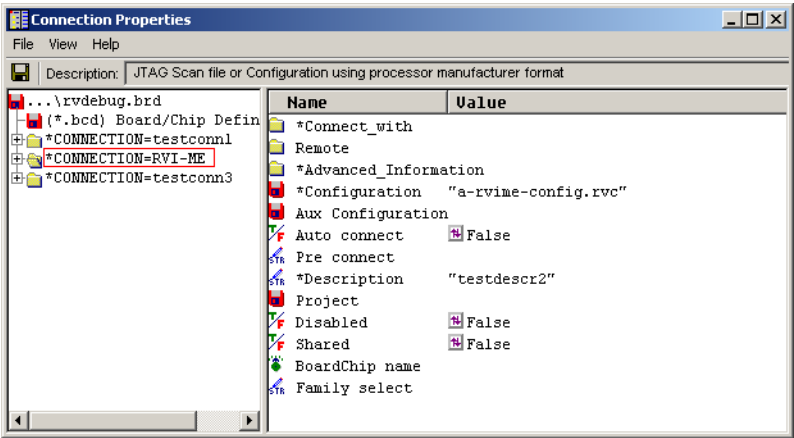


Figure 15-3 CONNECTION groups in the Connection Properties window

5. Right-click on the CONNECTION=RVI-ME entry, in the left pane.
6. Select **Make New...** from the context menu.
7. This displays the Group Type/Name selector dialog box, shown in Figure 14-3 on page 14-8.

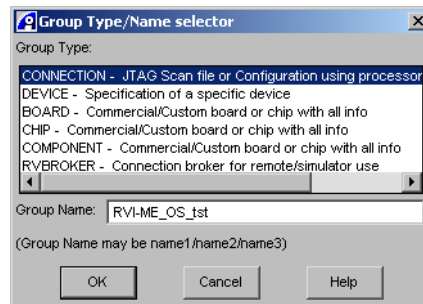


Figure 15-4 Specifying a new CONNECTION group

Leave the type of the new entry unchanged as CONNECTION.

In the Group Name data field change the name from RVI-ME to something suitable for your target, for example RVI-ME_OS_tst.

8. Click **OK** to confirm your settings and to close the Group Type/Name selector dialog box.

The new entry appears in the left pane of the Connection Properties window. It is automatically selected, and its details are displayed in the right pane. These details are the default for a new CONNECTION and you must change at least the Connect_with/Manufacturer, the Configuration filename, and target Description. The next steps explain how to make these changes.

9. In the right pane of the Connection Properties window, right-click on the Configuration entry and select **Edit as Filename** from the context menu.

The Enter New Filename dialog box is displayed to enable you to locate the required .rvc file, for example rvi_ARM_2.rvc.

10. Click **Save** to confirm your entries and to close the Enter New Filename dialog box.

The new pathname is displayed in the right pane.

11. In the right pane of the Connection Properties window, right-click on the Description field, and select **Edit Value** from the context menu.

Type RVI-ME to RTOS test board in the entry area and press Enter.

This is the description displayed in the Connection Control window and Connection Properties window to identify the new target.

12. In the right pane of the Connection Properties window, right-click on the Connect_with entry and select **Explore** from the context menu.

13. In the right pane of the Connection Properties window, right-click on the Manufacturer entry and select the required connection type from the context menu, that is **ARM-ARM-USB**.

If you do not specify this setting, the new connection appears in the Connection Control window but, when you try to connect, RealView Debugger prompts for the connection type.

14. Select **File** → **Save and Close** to save your changes and close the Connection Properties window.

Your new RVI-ME target is now displayed in the Connection Control window, shown in Figure 15-5.

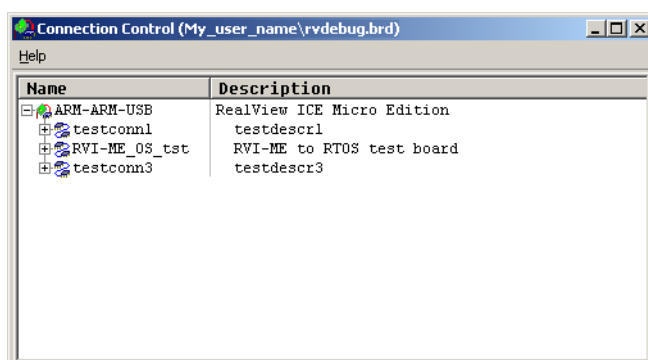


Figure 15-5 New connection in the Connection Control window

Configuring an RVI/RVI-ME interface unit

Ensure that the RVI/RVI-ME unit is configured as described in the chapter on configuring custom connections in Chapter 14 *Configuring Custom Connections* before you continue with this section.

Remember the following when specifying settings for your hardware:

- Autoconfiguring the RVI/RVI-ME unit does have side-effects and might be intrusive. Where this is not acceptable, you must configure manually.
- Be aware that clicking the option **Reset on Connect** might interfere with the initialization sequence of your application or target hardware.
- The RVI/RVI-ME scan chain configuration lists devices in ascending order of TAP ID.

Note

For more information on the tasks described here, and for full details on how to configure targets and create new connections, see Chapter 13 *Configuring Custom Targets* and Chapter 14 *Configuring Custom Connections*.

Now you must configure RTOS support for the new connection:

- If you have an RTOS-specific .bcd file, you can enable RTOS support on your target by referencing the .bcd file from your board file. Do this using the BoardChip_name entry as described in *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file*.
- If you do not have an RTOS-specific .bcd file, configure the RTOS on your target as described in your RTOS documentation coupled with the information in *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file* on page 15-14.

15.2.3 Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file

To configure the new connection to reference a .bcd file:

1. Ensure that you can see the new connection in the Connection Control window, shown in Figure 15-5 on page 15-11.
2. Right-click on the new connection and select **Connection Properties...** from the context menu to display the Connection Properties window. Use this to configure your board file.
3. Expand the (*.bcd) Board/Chip Definitions entry, in the left pane, to see a full list of all .bcd files detected by RealView Debugger. This includes the vendor-supplied file copied earlier (see *Enabling RTOS support* on page 15-7).
4. Right-click on the entry BoardChip_name, in the right pane, and select the required entry from the list, for example **Rtos_Trigon_RSD_NonStop**. Select <More...> to see the full list.

Your window looks like Figure 15-6 on page 15-13.

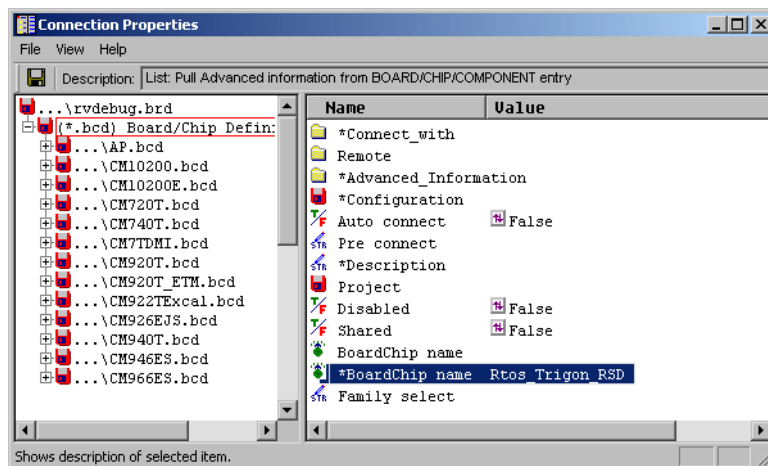


Figure 15-6 Referencing the RTOS .bcd file

5. If you want to reference entries from other .bcd files from this connection, do this now. Right-click on the entry BoardChip_name, in the right pane, and select the required group from the list, for example AP.

———— **Note** ————

See *Referencing non-RTOS .bcd files* on page 15-20 for notes on working with multiple .bcd files.

6. Select **File** → **Save and Close** to save your changes to the board file and close the Connection Properties window.
7. Connect to your RTOS-enabled target and load an image, as described in *Connecting to the target and loading an image* on page 15-22.

———— **Note** ————

RealView Debugger provides great flexibility in how to configure configuration settings so that you can control your debug target and any custom hardware that you are using. Where settings conflict, priority depends on the type of setting, whether it has changed from the default, and its location in the configuration hierarchy. For example, connection mode settings in the board file take priority over the same setting in any linked .bcd files. See *Managing configuration settings* on page 15-21 for details.

There are descriptions of the general layout and controls of the RealView Debugger settings windows, including the Connection Properties window, in the RealView Debugger online help topic *Changing Settings*.

For full details on the tasks in this example, and how to configure RealView Debugger targets for first use, see Chapter 13 *Configuring Custom Targets*.

15.2.4 Configuring an RTOS-enabled connection without a vendor-supplied .bcd file

If you do not have a vendor-supplied .bcd file, you must configure RTOS operation for your new connection in your board file. For ARM architecture-based targets, RTOS operation is controlled by settings groups in the Advanced_Information block:

- Advanced_Information - Default - RTOS
- Advanced_Information - Default - ARM_config.

Note

Do not configure the board file when the debugger is connected to a target.

RTOS configuration options

To configure RTOS settings for the new connection:

1. Ensure that you can see the new connection in the Connection Control window, shown in Figure 15-5 on page 15-11.
2. Right-click on the new connection and select **Connection Properties...** from the context menu to display the Connection Properties window. Use this to configure your board file.
3. Double-click on the Advanced_Information group, in the right pane, to expand the Advanced_Information block.
4. Double-click on the Default group, in the right pane, to see the RTOS and ARM_config groups.
5. Double-click on the RTOS group, in the right pane, to see the RTOS configuration settings, shown in the example in Figure 15-7 on page 15-15.

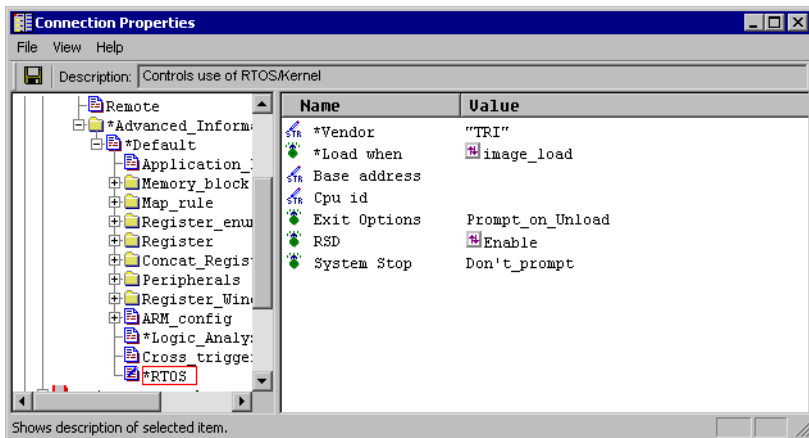


Figure 15-7 RTOS group in the Connection Properties window

As shown in Figure 15-7, configure RTOS operation in your board file using the RTOS group:

Vendor This three letter value identifies the RTOS plugin, that is the *.dll file supplied by your vendor.

Load_when Defines when RealView Debugger loads the RTOS plugin:

- load the plugin on connection, with Load_when set to **connect**
- wait until an RTOS image is loaded, with Load_when set to **image_load**.

The RTOS features of the debugger are not enabled until the plugin is loaded and has found the RTOS on your target.

When the plugin is loaded, it immediately checks for the presence of the RTOS. If loaded, the plugin also checks when you load an image. This means that you might have to run the image startup code to enable RTOS features in the debugger.

Base_address

Defines a base address, overriding the default address used to locate the RTOS data structures. See your RTOS documentation for details.

Cpu_id

Specifies a CPU identifier so that you can associate the OS-aware connection to a specific CPU. See your RTOS documentation for details.

Exit_Options

Defines how RTOS awareness is disabled. Use the context menu to specify the action to take when an image is unloaded or when you disconnect. You can also specify a prompt.

RSD

Controls whether RealView Debugger enables or disables RSD. This setting is only relevant if your debug target can support RSD.

If you load an image that can be debugged using RSD, set this to **Disable** to prevent RSD starting automatically. You can then start RSD from the Resource Viewer window or the Process Control pane (see *Using the Resource Viewer window* on page 15-41 and *Working with OS-aware images in the Process Control pane* on page 15-34 for details).

System_Stop

Use this setting to specify how RealView Debugger responds to a processor stop request when running in RSD mode.

In some cases, it is important that the processor does not stop. This setting enables you to specify this behavior, use:

- **Never** to disable all actions that might stop the processor.
- **Prompt** to request confirmation before stopping the processor.
- **Don't_prompt** to stop the processor. This is the default.

Consider the following when specifying these settings:

- Set Load_when to **connect** or **image_load** for RSD mode, depending on whether the Debug Agent is built into the RTOS or the image.

This is important when you are connecting to a running target. When RealView Debugger connects, the Debug Agent might be found but symbols are not yet loaded and so the OS marker shows RSD (PENDING SYMBOLS). The Debug Agent might communicate, to the debugger, all the information necessary to start RSD. In this case, RealView Debugger switches to RSD mode immediately, and you can read memory while the target is running.

Otherwise, contact is made with the Debug Agent but RSD is not fully operational.

See *OS marker in the Process tab* on page 15-34 for details.

- In some cases settings in the RTOS group might conflict and so are ignored by RealView Debugger:

Exit_Options

RealView Debugger might ignore any of the ***_on_Unload** settings, if the image that is being unloaded has no relevance for the underlying RTOS, or if RTOS support has not been initialized.

System_Stop

These settings might conflict with the connect or disconnect mode configuration settings for the current target. This means that your target might stop on connect or disconnect even where you have specified **Never** or **Prompt** for this RTOS setting.

Configure the following settings in your .brd file, or the vendor-supplied .bcd file, to avoid this problem specify:

- connection status by setting Advanced_Information - Default - Connect_mode
- disconnect status by setting Advanced_Information - Default - Disconnect_mode.

See *Connect and disconnect configuration options* on page 15-18 for more details on these settings.

Remember to select **File** → **Save and Close** to save your changes to the board file and close the Connection Properties window.

ARM configuration options

For ARM architecture-based targets, you must also specify the ARM_config settings group in the Advanced_Information block in your board file, shown in Figure 15-8.

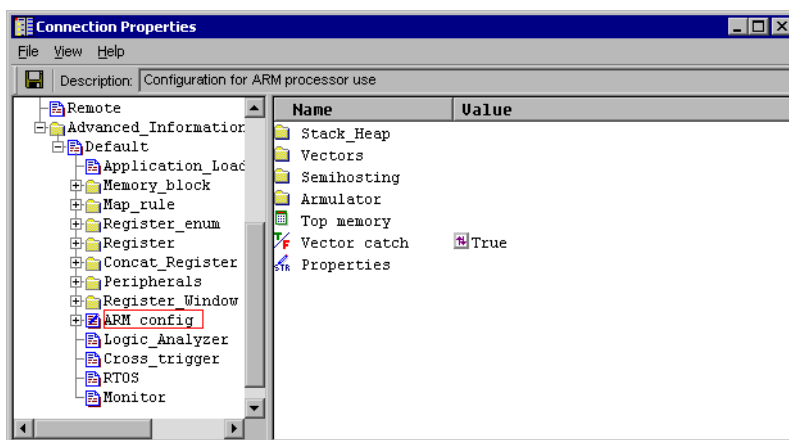


Figure 15-8 ARM_config group in the Connection Properties window

For this group, configure the following settings:

1. Double-click on the Vectors group so that the contents are displayed in the right pane.

2. Set Undefined to False.
To implement RSD breakpoints, the Debug Agent must catch undefined exceptions. Ensure that you configure this setting as described.
3. Set D_Abort to False.
4. Define other ARM_config settings as required by your application.
5. Select **File** → **Save and Close** to save your changes to the board file and close the Connection Properties window.

Note

Remember, it is not necessary to make any changes to settings in your board file where your RTOS vendor has supplied an appropriate .bcd file, see *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file* on page 15-12.

Connect and disconnect configuration options

If you want to specify how RealView Debugger connects to (or disconnects from) a target processor, you can configure this in your board file. These definitions are contained in the Advanced_Information block.

The configuration settings Connect_mode and Disconnect_mode are a special case when used to configure a debug target:

- If a prompt is specified in your board file, or in any .bcd file linked to the connection, it takes priority over any other user-defined setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.
- If a (non-prompt) user-defined setting is specified in your board file and in any .bcd file linked to the connection, the board file setting takes priority.
- A blank entry in the top-level Advanced_Information block ensures that any setting in a linked Board/Chip definition file is used instead. This might be important if you are using a vendor-supplied .bcd file to enable RTOS awareness (see *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file* on page 15-12 for details).

To configure connect and disconnect behavior for the new connection:

1. Ensure that you can see the new connection in the Connection Control window, shown in Figure 15-5 on page 15-11.

2. Right-click on the new connection and select **Connection Properties...** from the context menu to display the Connection Properties window. Use this to configure your board file.
3. Double-click on the Advanced_Information group, in the right pane, to expand the Advanced_Information block.
4. Double-click on the Default group, in the right pane, to see the Connect_mode and Disconnect_mode settings, shown in the example in Figure 15-9.

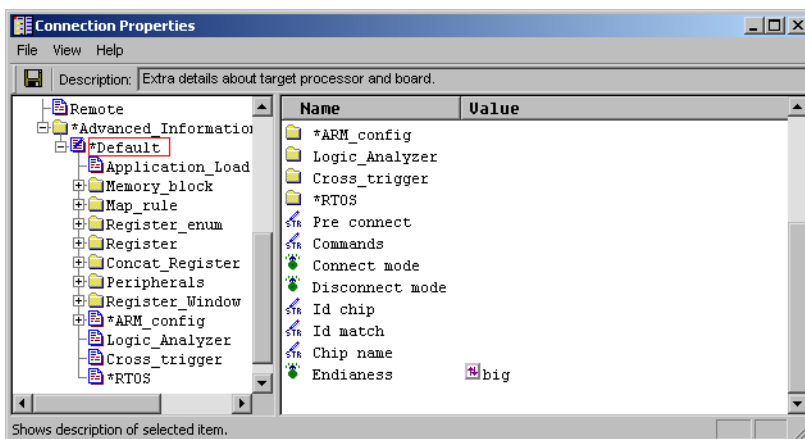


Figure 15-9 Default group in the Connection Properties window

As shown in Figure 15-9, configure connection behavior in your board file using the settings:

- Connect_mode
- Disconnect_mode.

Right-click on each setting to see the options available to you. These options are fixed and so might include options that are not supported by your target vehicle. If you specify such an option, the debugger prompts you to select an appropriate mode when you try to connect or disconnect.

For example, if you do not want to stop the target but still want to enable RSD mode, set Connect_mode to **no_reset_and_no_stop** from the context menu.

Remember to select **File → Save and Close** to save any changes to the board file and close the Connection Properties window.

Note

The connect (or disconnect) mode that is actually used depends on the target hardware, the target vehicle, and the associated interface software that manages the connection. If you are using RVI/RVI-ME, the unit configuration determines the connect mode and makes the connection. Therefore, the unit configuration might override any settings that you specify in your board file.

For more details on how RealView Debugger connects to, and disconnects from, a target, see the chapter describing connecting to targets in Chapter 13 *Configuring Custom Targets*.

Referencing non-RTOS .bcd files

You can reference other .bcd files from this RTOS-enabled connection in the usual way:

1. Ensure that you can see the new connection in the Connection Control window, shown in Figure 15-5 on page 15-11.
2. Right-click on the new connection and select **Connection Properties...** from the context menu to display the Connection Properties window.
3. Expand the (*.bcd) Board/Chip Definitions entry, in the left pane, to see a full list of all .bcd files detected by RealView Debugger.
4. Right-click on the entry BoardChip_name, in the right pane, and select the required group from the list, for example **AP**.
5. Select **File** → **Save and Close** to save your changes to the board file and close the Connection Properties window.
6. Connect to your RTOS-enabled target and load an image, as described in *Connecting to the target and loading an image* on page 15-22.

When referencing .bcd files from an RTOS-enabled connection remember:

- Configuration settings define a hierarchy, starting from the general connection-level and becoming more specific, through whole chips to component modules on a chip. Where settings conflict, priority depends on the type of setting, whether it has changed from the default, and its location in the configuration hierarchy. For example, if you change any ARM_config settings from their defaults in the supplied board file, these take priority over the same setting in any linked .bcd files. See *Managing configuration settings* on page 15-21 for more details.

- It is recommended that you do not edit any settings in a supplied .bcd file in case these change in a future release of RealView Debugger, or if they are updated by your RTOS vendor. If required, define custom files by creating new target descriptions as explained in chapter Chapter 13 *Configuring Custom Targets*.
- Configuration settings defined as part of a project take precedence. Ensure that project settings do not conflict with target configuration settings. See *RealView Developer Kit v2.2 Project Management User Guide* for details.

For full details on the tasks in this example, and how to configure RealView Debugger targets for first use, see Chapter 13 *Configuring Custom Targets*.

15.2.5 Managing configuration settings

RealView Debugger provides great flexibility in how to configure configuration settings so that you can control your debug target and any custom hardware that you are using. This means that some settings can be defined in the top-level board file so that they apply to a class of connections, for example CONNECTION=RVI-ME_OS_tst, or on a per-board basis using groups in one or more linked .bcd files, for example AP.bcd or Rtos_Trigon_RSD_NonStop.bcd.

————— Note —————

To avoid conflicts between settings when you link multiple target description groups in .bcd files, follow the guidelines given in Chapter 13 *Configuring Custom Targets*.

To ensure that settings defined in one or more linked .bcd file are used to assemble the target configuration, do not change the default settings contained in the target connection group. For example, if you specify top_of_memory in a linked .bcd file, you must check that the same entry is blank (the default) in the top-level board file:

1. Select **Target** → **Connection Properties...** to display the Connection Properties window.
2. Expand the following entries in turn:
 - a. CONNECTION=RVI-ME_OS_tst
 - b. Advanced_Information
 - c. Default
 - d. ARM_config
3. Ensure that Top_memory is blank.

If the setting contains an entry, right-click to display the context menu. Because the setting has been configured, the menu now offers more options. Select **Reset to Empty** to create a blank setting.

15.3 Connecting to the target and loading an image

You connect to an RTOS target in the same way as non-RTOS targets, for example using the Connection Control window. This section describes how to connect to your target and load an image. It contains the following sections:

- *Before connecting*
- *Connecting from the Code window*
- *Connecting to a running target* on page 15-23
- *RTOS Exit Options* on page 15-24
- *Interrupts when loading an image* on page 15-25
- *Resetting OS state* on page 15-25
- *Loading from the command line* on page 15-26.

15.3.1 Before connecting

Ensure that you:

- compile your RTOS image with debug symbols enabled so that the debugger can find the data structures it requires for HSD. If you are in RSD mode, this might not be necessary.
- install your RTOS plugins as described in *Enabling RTOS support* on page 15-7.
- create a new RTOS connection as described in *Creating a new RTOS-enabled connection* on page 15-8.
- configure your RTOS-enabled connection as described in either:
 - *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file* on page 15-12
 - *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file* on page 15-14.

15.3.2 Connecting from the Code window

To connect to an RTOS target:

1. Start RealView Debugger.
2. Use the Connection Control window to connect to the target using the RTOS-enabled connection.
3. Select **Target** → **Load Image...** to load the image.

If you have several executable files to load, use the ADDFILE and RELOAD commands.

15.3.3 Connecting to a running target

Depending on your target, connecting to a running target results in different startup conditions, either:

- RSD is enabled and contact with the Debug Agent is established. You can start working with threads because the Debug Agent has communicated all the necessary information to the debugger to start RSD.

Note

In this case, you must also load symbols to start debugging. It is not necessary to load the RTOS symbols, because the application symbols are sufficient.

- RSD is enabled and contact with the Debug Agent is established. However, the information necessary to start RSD is part of the RTOS symbols. In this case, you must load symbols before you can debug your image.

If you want to connect to a running target, or disconnect from a target without stopping the application, use the configuration settings `Connect_mode` and `Disconnect_mode`, as described in *Connect and disconnect configuration options* on page 15-18.

If you have configured your connection to reference a vendor-supplied `.bcd` file that defines nonstop running, the connection options are defined by these settings and take precedence over any settings elsewhere. In this case, it is not necessary to specify the connection mode in your board file.

Specifying connect mode

When you connect to a running target, you can override any settings in your board file, or in any referenced vendor-supplied `.bcd` file, to specify the connection mode used. This option is also available when you disconnect from a target without stopping (see *Specifying disconnect mode* on page 15-24 for details). To specify the connect mode, use the context menu in the Connection Control window:

1. Start RealView Debugger.
2. Display the Connection Control window to view the RTOS-enabled connection that is running your application.
3. Right-click on the connection entry in the Connection Control window and select **Connect (Defining Mode)...** from the **Connection** context menu.
4. Select **No Reset** and **No Stop** (default) from the options.
5. Click **OK** to close the Connect Mode selection box.

6. Where the OS marker shows RSD (PENDING SYMBOLS), select **Target → Load Image...** to load the symbols. In the Load File to Target dialog box, remember to:
 - check the **Symbols Only** option
 - uncheck **Auto-Set PC**
 - uncheck **Set PC to Entry point**.

Specifying disconnect mode

To disconnect from a target without stopping the application, use either the connection options available from the Code window **File** menu or the context menu in the Connection Control window:

1. Start RealView Debugger. Configure RTOS support and load the multithreaded image.
2. Start the image running until you reach a point where threads are being rescheduled.
3. Select **Target → Disconnect (Defining mode)** to define the disconnection mode. You can also right-click on the connection entry in the Connection Control window and select **Disconnect (Defining Mode)...** from the **Disconnection** context menu.
4. Select the required option, for example As-is without Debug from the options.
5. Click **OK** to close the Disconnect Mode selection box.

You can now exit RealView Debugger but leave your debug target in its current state, for example running your RTOS application.

See Chapter 13 *Configuring Custom Targets* for full details on connect, and disconnect, modes.

15.3.4 RTOS Exit Options

The RTOS group configuration settings define how RTOS awareness is disabled. You can specify the action to take when an image is unloaded or when you disconnect. Ensure that these do not conflict with the connect or disconnect mode specified for your target (see *Connect and disconnect configuration options* on page 15-18 for details).

When you unload (or reload) an image, for example using the **Process** tab context menus, the Exit_Options setting decides how to disable RTOS awareness. If this is set to **Prompt_on_Unload**, the default setting, RealView Debugger displays a selection box to enable you to specify the exit conditions, shown in Figure 15-10 on page 15-25.

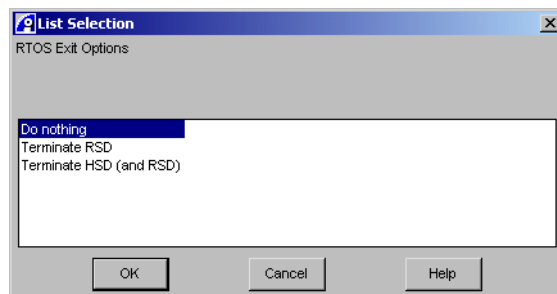


Figure 15-10 RTOS Exit Options selection box

This selection box offers:

Do nothing Select this to maintain the current state.

Terminate RSD

Select this to disable RSD and end communication with the Debug Agent.

Terminate HSD (and RSD)

Select this to end communication with the Debug Agent and unload the RTOS plugin, that is the *.dll file.

Select the required state and then click **OK** to close the selection box.

If you click **Cancel**, this is the same as choosing **Do nothing** → **OK**.

15.3.5 Interrupts when loading an image

When you load an image to your target, ensure that all interrupts are reset. If interrupts are not reset when the image executes, either by a STEP or G0 command, an IRQ might occur associated with the previous image that could cause the current image to run incorrectly.

15.3.6 Resetting OS state

The RTOS plugin samples the OS state to determine the current state of the RTOS kernel, for example initialized, nearly initialized, or uninitialized. If you load and run an image on an RTOS-enabled target, stop, and then immediately reload, without resetting the OS state to its initial value, HSD is enabled and any resources shown are those of the previous image execution.

To ensure that this does not happen, always reset the OS state to its uninitialized value each time the RTOS is reloaded.

15.3.7 Loading from the command line

You can start RealView Debugger from the command line and specify an image to load automatically. To do this:

1. Ensure that your workspace specifies an open connection, so that the debugger automatically connects on startup.

If you are not connected to your debug target before starting RealView Debugger, loading an image from the command line starts the debugger and then displays a prompt box for you to complete the connection. When you are successfully connected, the image is loaded.

Alternatively, use the -INIT command line option to specify a target connection.
2. Provide the executable filename and any required arguments on the command line.

15.4 Associating threads with views

When HSD or RSD is fully operational, you can start to work with threads in the RealView Debugger Code window. This is described in the following sections:

- *Attaching and unattaching windows*
- *The current thread* on page 15-28
- *Using the Cycle Threads button* on page 15-28
- *Working with the thread list* on page 15-30.

You can also work with threads in the Process Control pane, see *Working with OS-aware images in the Process Control pane* on page 15-34 for details.

15.4.1 Attaching and unattaching windows

If you are working with an RTOS-enabled connection, you can attach Code windows to threads.

The windows attachment status is displayed in the Code window title bar, after the name of the target:

[Unattached] Specifies that the Code window is not attached to a connection, that is a debug target board or a specified processor on a multiprocessor board.

By default, unattached Code windows display details of the *current thread*, that is the thread that was most recently running on the target when the target stops.

[Board] Specifies that the Code window is attached to a connection, that is a debug target board or a specified processor on a multiprocessor board. This window displays details of the current thread on that target, if available.

<blank> If the title bar contains no attachment details then this window is attached to a specified thread, that is it is always associated with this thread.

————— Note —————

If you use **View → New Code Window** to create a new Code window, it inherits its attachment from the calling window.

When working with threads, you can change the attachment of your Code window using the *thread list*, see *Working with the thread list* on page 15-30 for details.

15.4.2 The current thread

When working with a multithreaded application, the current thread is initially set to the thread that was running on the processor when it stopped. If you are working with an unattached Code window, this shows details about the current thread.

When the current thread changes, for example when you stop the target with a different thread active, the **Cmd** tab of the Output pane displays details of the new current thread. This includes the thread number in decimal and the thread name, if it is available.

Initially in RSD, the current thread is undefined and so RealView Debugger designates a thread at random to be the current thread. However, you can change the current thread using the **Cycle Threads** button, see *Using the Cycle Threads button* for details.

Using CLI commands

If you are working in an unattached Code window, the current thread defines the scope of many CLI commands. If you are working in an attached window, the scope of CLI commands is defined by the attached thread.

You can use CLI commands to work with threads, for example:

`print @r1` Print the value of the thread that was current when the processor stopped.

`thread,next` Change the current thread.

See *RealView Developer Kit v2.2 Command Line Reference* for a full description of the `THREAD` command.

15.4.3 Using the Cycle Threads button



When HSD or RSD is fully operational, this enables the **Cycle Threads** button and drop-down arrow on the Connect toolbar in the Code window:

- Use the **Cycle Threads** button to view thread details and change the current thread, see *Viewing thread details* on page 15-29 for details.
- Use the **Cycle Threads** button drop-down to access the thread list where you can change your thread view and windows attachment, see *Working with the thread list* on page 15-30 for details.

————— Note —————

For HSD, the thread list is only available when the processor is stopped.

Viewing thread details



Use the **Cycle Threads** button to view each of the threads on the system in turn. Click the **Cycle Threads** button to cycle an unattached window through the threads so that it displays details about a new thread. This changes the current thread and updates your code view. The new current thread appears in the Code window title bar and the Color Box changes color.

You can only cycle through the threads in this way in a Code window that is not attached to a thread. If your Code window is attached to a thread and you try to cycle threads in this way, a dialog box appears:

Window attached. Do you want to detach first?

Click **Yes** to unattach the window and change the thread view. Click **No** to abort the action and leave the thread view unchanged.

———— Note ————

You can use the **Cycle Threads** button to cycle through the thread list in a Code window that is attached to a *connection* without changing the windows attachment.

If you click the **Cycle Threads** button to change the current thread:

- the **Cmd** tab of the Output pane displays the thread, next command
- the **Log** tab of the Output pane displays details of the new current thread, that is the thread number in decimal and the thread name.

You can also change the current thread using the **Thread** tab in the Process Control pane, see *Using the Thread tab* on page 15-38 for details.

15.4.4 Working with the thread list

The thread list on the **Cycle Threads** button is available:

- for a processor running in HSD mode, when that processor is stopped
- for a processor running in RSD mode.



Click on the **Cycle Threads** button drop-down arrow to cause RealView Debugger to fetch the list of threads from the target and display a summary, as in the example in Figure 15-11.

Note

The number of threads a Debug Agent can handle is defined by your RTOS vendor. When the thread buffer is full, no threads can be displayed.

Attach Window to a Thread				
0x00004124	System Timer Thread 0	SUSP	0x00000000	
0x000038b8	thread 0	1 SLEEP	0x00000000	
0x00003948	thread 1	16 READY	0x00000000	
0x000039d8	thread 2	16 READY	0x00000000	
*0x00003a68	thread 3	8 READY	0x00000000	
0x00003af8	thread 4	8 READY	0x00000000	
0x00003b88	thread 5	4 SUS_EV	0x00000000	

Figure 15-11 Example thread list

The first option on this menu is **Attach Window to a Thread**. Use this to control windows attachment, see *Attaching windows to threads* on page 15-32 for details.

Below the menu spacer is a snapshot of the threads running on the target when the request was made. In the example in Figure 15-11, the fields shown (from left to right) for each thread are the:

- address of the thread control block
- name of the thread
- priority of the thread
- status of the thread (for example, ready, sleeping, or suspended event)
- thread suspend flags.

Click on a new thread, in the thread list, to change the code view so that it displays the registers, variables, and code for that thread:

- If you click on a new thread in an unattached Code window, it becomes attached to the thread automatically. The thread details appear in the Code window title bar and the Color Box changes color.
If you change the thread view in this way, other unattached windows are not affected, that is they remain unattached and continue to show the current thread.
- If you click on a new thread in a Code window that is attached to a connection, it becomes attached to the thread automatically.
- If you click on a new thread in a Code window that is attached to a thread, it becomes attached to the specified thread.

In this release, the Debug Agent handles up to 64 threads. Where the thread list does not show the full details, use the thread selection box to see all the threads detected on the system (see *Using the thread selection box* on page 15-33 for details).

Current thread

As described in *The current thread* on page 15-28, RealView Debugger designates a thread to be the current thread when you are in RSD. In Figure 15-11 on page 15-30, the asterisk (*) shows the current thread. Because the Code window is unattached, any thread-specific CLI commands you submit operate on this thread.

In a Code window that is attached to a thread, shown in Figure 15-12 on page 15-32, the asterisk shows the current thread but any CLI commands operate on the attached thread, marked by a check mark.

See *Working with OS-aware images in the Process Control pane* on page 15-34 for more details on viewing threads.

Captive threads

The thread list shows:

- All threads on the system that can be captured by RealView Debugger, that is they can be brought under debugger control. These are called *captive threads*.
- Special threads, scheduled along with other tasks in the system, that cannot be captured, that is they are not under the control of RealView Debugger. These *non-captive threads* are grayed out.

Figure 15-11 on page 15-30 shows two grayed threads (see *Special threads in the Thread tab* on page 15-40):

- Debug Agent
- *IMP Comms Target Manager* (ICTM). Part of the Debug Agent, this handles communications between the Debug Agent and the target.

These threads are essential to the operation of RSD and are grayed out to show that they are not available to RealView Debugger. Which threads are grayed out depends on your target.

Attaching windows to threads



Click on the **Cycle Threads** button drop-down arrow to display the thread list, shown in Figure 15-11 on page 15-30. The first menu item is **Attach Window to a Thread**. Select this option to attach the Code window to the current thread. Select it again to unattach an attached Code window.

If you display the thread list from an unattached Code window, click on a thread to change the thread view and attach the window automatically. The thread details appear in the Code window title bar and the Color Box changes color.

If you display the thread list from a Code window that is attached to a thread, the first menu item, **Attach Window to a Thread**, is ticked. The attached thread is also marked by a check mark, shown in Figure 15-12.

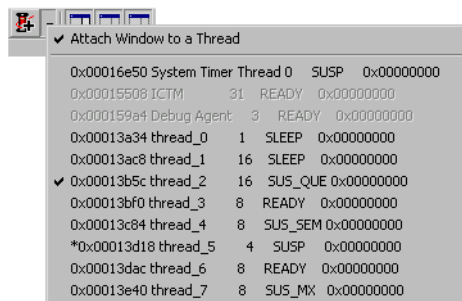


Figure 15-12 Example thread list in an attached window

Note

If you click on a thread in the thread list, it does not become the current thread. This changes the thread view and attaches the Code window. However, if you click the **Cycle Threads** button to change the thread, the next thread in the thread list becomes the current thread.

You can also change windows attachment using the **Thread** tab, see *Using the Thread tab* on page 15-38 for details.

Using the thread selection box

The thread list might contain a large number of threads. In this case, the list is shortened and the menu contains the option **<More Threads...>** to display the full contents. Select this option to display the thread selection box to see a full thread list, shown in Figure 15-13.

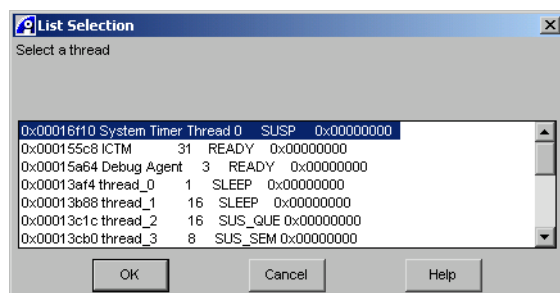


Figure 15-13 Thread selection box

The thread selection box shows a full list of threads. Use the selection box in the same way as the thread list, for example, to select a thread and so change the thread view.

15.5 Working with OS-aware images in the Process Control pane

The Process Control pane shows details about each connection known to RealView Debugger. When the RTOS has been detected, the pane contains the following tabs:

- | | |
|----------------|--|
| Process | Use the Process tab to see the processor details, project details, and information about any image(s) loaded onto the debug target, for example: <ul style="list-style-type: none"> • image name • image resources, including DLLs • how the image was loaded • load parameters • associated files • execution state. |
| Map | If you are working with a suitable target you can enable memory mapping and then configure the memory using the Map tab. See Chapter 6 <i>Memory Mapping</i> for more details. |
| Thread | This displays RTOS-specific information about the threads that are configured on the target. |

This section describes:

- *OS marker in the Process tab*
- *Using the Thread tab* on page 15-38.

15.5.1 OS marker in the Process tab

Select **View** → **Process Control** to display the Process Control pane if it is not visible in your Code window. The following sections describe the OS marker details:

- *OS marker initial state* on page 15-35
- *OS marker with and an OS-aware image loaded* on page 15-35
- *OS marker for a running target with RDS fully operational* on page 15-36
- *OS marker status* on page 15-36
- *Context menu* on page 15-37.

OS marker initial state

If HSD or RSD is enabled, the **Process** tab contains the OS marker, shown in Figure 15-14. This is the initial state with no image loaded.

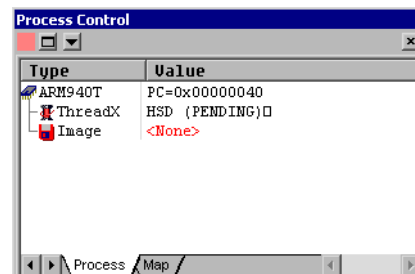


Figure 15-14 OS marker initial state in the Process Control pane (HSD)

In this state, neither the **Thread** tab nor the **Cycle Threads** button is available.

OS marker with and an OS-aware image loaded

With an RTOS-enabled image loaded (thr_demo.axf) but not running, the **Process** tab contains the OS marker, shown in Figure 15-15.

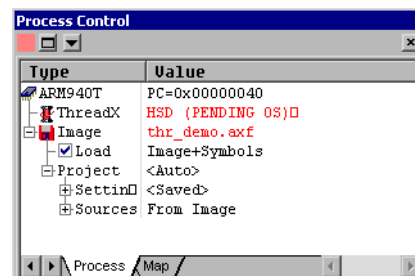


Figure 15-15 OS marker with loaded image in the Process Control pane (HSD)

Figure 15-15 shows a debug target where RealView Debugger has located the RTOS using the RTOS plugin. Although a suitable RTOS-enabled image has been loaded to the target, RealView Debugger has not detected that the RTOS has started.

In this state, neither the **Thread** tab nor the **Cycle Threads** button is available.

OS marker for a running target with RDS fully operational

Figure 15-16 shows a running target where RealView Debugger has detected that a suitable RTOS-enabled image has been loaded to the target and threads are being rescheduled.

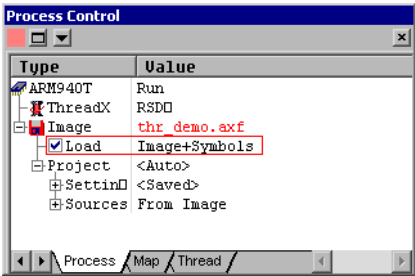


Figure 15-16 OS marker in the Process Control pane (RSD)

In this state, the **Thread** tab is available and the **Cycle Threads** button is enabled on the Connect toolbar.

OS marker status

Table 15-1 describes OS marker status in the Process Control pane.

Table 15-1 OS marker status in the Process Control pane

Status	Meaning
NOT INITIALIZED	HSD or RSD is enabled but the triggering event that loads the plugin, the *.dll file, has not occurred.
HSD (PENDING)	The plugin has been loaded and RealView Debugger is waiting for it to determine that the RTOS has been found.
HSD (PENDING OS)	The plugin has found the RTOS but the process is not yet in a state where threads are being rescheduled.
HSD	HSD is fully operational.
HSD (RUNNING)	The system is running when RSD is disabled. This means that no up-to-date information about the RTOS can be shown. This value is never shown when RSD is enabled.
RSD (PENDING DA)	RealView Debugger is waiting for notification that a Debug Agent has been found.
RSD	RSD is fully operational.

Table 15-1 OS marker status in the Process Control pane

Status	Meaning
HSD (RSD STOPPED)	RealView Debugger has changed from RSD to HSD debugging mode. This is usually because an HSD breakpoint or processor stop command has been performed. HSD is now available.
HSD (RSD INACTIVE)	RSD was operational or pending but is now disabled. This occurs only by a user request. HSD is now available.
HSD (RSD DEAD)	RSD was operational but is now disabled. This is usually because the Debug Agent is not responding to commands or the RTOS has crashed. HSD is now available.
RSD DEAD	RSD was operational but is now disabled. The debug target is still running. HSD is not available.
RSD (PENDING SYMBOLS)	RSD is operational, contact to the Debug Agent has been established but no symbols have been loaded. This means that RSD only works partially until the symbol table is loaded. This usually occurs after connecting to a debug target that is already running the Debug Agent.

The OS marker is usually shown in black text in the **Process** tab. Where the marker is shown in red, either RTOS support is not ready, as shown in Figure 15-15 on page 15-35, or there has been an error.

Context menu

Right-click on the OS marker to see the context menu where you can control RSD:

Disable RSD Depending on the current mode, this option enables or disables RSD. For more details on how to control RSD, see the **RSD** menu described in *Resource Viewer window interface components* on page 15-42.

The initial state depends on the RSD setting in the RTOS group in the board file, or .bcd file where available. See *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file* on page 15-14 for details.

Stop Target Processor

Stops the target processor and suspends RSD. Depending on your configuration settings, click the **Go** button to start execution and restart RSD.

Properties Select this option to see details about the Debug Agent. This includes the status of the RSD module, settings as specified in your board file (or .bcd file where available), and RSD breakpoints. For an example, see *More on breakpoints* on page 15-48.

15.5.2 Using the Thread tab

The **Thread** tab in the Process Control pane displays RTOS-specific information about the threads that are configured on the target:

- In HSD mode, the **Thread** tab shows the thread state when the processor is stopped. When you start a processor that runs in HSD mode, the thread details are cleared from the tab.
- In RSD mode, the **Thread** tab shows a snapshot of the last known state of the system.

Select **View** → **Process Control** to display the Process Control pane if it is not visible in your Code window. Click on the **Thread** tab.

———— **Note** ————

To display the Thread tab quickly if the Process Control pane is not visible, select **View** → **Threads tab**.

Expand the tree to see each configured thread and the associated summary information, shown in Figure 15-17.

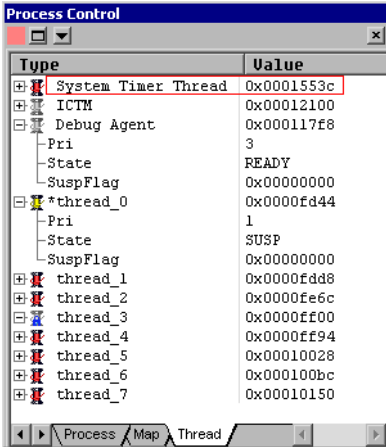






Figure 15-17 Thread tab in the Process Control pane

In this example, the Process Control pane is floating and so the pane title bar reflects the title bar of the calling Code window. Figure 15-17 on page 15-38 shows that the Code window is attached to thread 3.

Thread icons

Each thread is identified by an icon:

-  A red icon indicates a thread that is currently not captive.
-  A blue padlock indicates that the Code window is attached to this thread.
-  A gray icon indicates a thread that is not under the control of RealView Debugger and so cannot become captive.
-  When you are working in RSD mode, a yellow icon indicates that a thread is captive, for example after hitting a breakpoint.

The padlock icon is shown if the Code window is attached to the thread, regardless of its captive state.

In HSD mode, a yellow icon indicates the thread that was running when the target stopped.

The asterisk (*) shows the current thread.

Changing the current thread and attachment

You can change the current thread and attachment status from the Process Control pane. Right-click on the thread name to display a context menu containing the options:

Update All Used for RSD, select this option to update the system snapshot. This has no effect in HSD but is not grayed out.

Make Current

Make this thread the current thread (see *The current thread* on page 15-28).

Attach Window to

Attach the Code window to this thread (see *Attaching windows to threads* on page 15-32).

Special threads in the Thread tab

In this example, the **Thread** tab includes special threads that are visible but are not available to you:

ICTM Part of the Debug Agent, the *IMP Comms Target Manager* handles communications between the Debug Agent and the target.

Debug Agent

The main part of the Debug Agent runs as a thread under the target RTOS. This passes thread-level commands to RealView Debugger.

See *Debug Agent* on page 15-4 for details.

15.6 Using the Resource Viewer window

The Resource Viewer window gives you visualization of RTOS resources, for example the thread list, and RTOS objects such as mutexes, queues, semaphores, memory block pools, and memory byte pools.

This section describes how to use the Resource Viewer window, and includes:

- *Displaying the Resource Viewer window*
- *Working with RTOS resources* on page 15-42
- *Resource Viewer window interface components* on page 15-42
- *Using Action context menus* on page 15-45
- *Interaction of RTOS resources and RealView Debugger* on page 15-46.

15.6.1 Displaying the Resource Viewer window

To display this window, shown in Figure 15-18, select **View** → **Resource Viewer Window** from the Code window main menu.

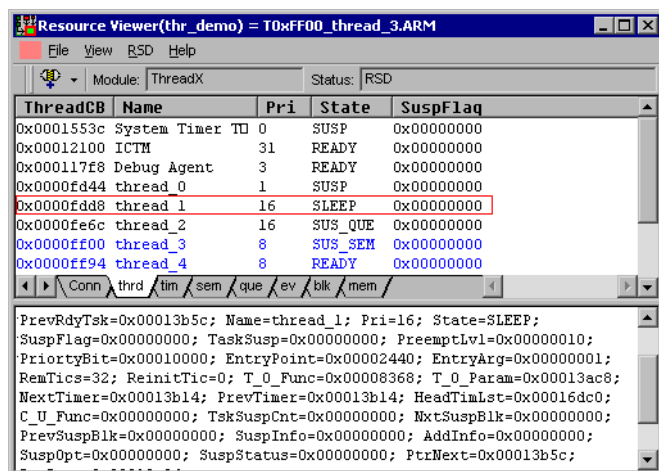


Figure 15-18 Resource Viewer showing thread details

The Resource Viewer window title bar reflects the title bar of the calling Code window. In this example, the Code window is attached to thread 3.

15.6.2 Working with RTOS resources

The tabbed pane at the top of the Resource Viewer window contains the Resources list. This displays all the resources available to RealView Debugger. Where no RTOS has been detected, the Resources list contains only the **Conn** tab showing the connection.

———— **Note** ————

If you are in HSD mode, you must stop execution to view your RTOS resources.

Click on a connection in the Conn tab and select **View → Display Details** to see a short description in the Details area in the window. You can also display details about an item by double-clicking on the entry in the Resources list.

15.6.3 Resource Viewer window interface components

This section describes the RTOS-specific features of the Resource Viewer window:

File menu This menu contains:

Close Window

Closes the Resource Viewer window.

View menu This menu contains:

Update List

Updates the items displayed in the Resources list.

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

Display Details

Displays details about a selected entry in the Resources list. A short description is shown in the Details area in the window. You can also display details about an item by double-clicking on the entry in the Resources list.

Display Details as Property

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

Select this option to change the display format while the window is open. Close the Resource Viewer window to restore the default, that is a description is shown in the Details area.

Display List in Log Area

Not available in this release.

Clear Log

Clears messages and information displayed in the Details area.

Auto Update Details on Stop

Automatically updates the Details area when any image running on the connection stops. This gives you information about the state of the connection when the process terminated.

Auto Update

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

———— **Note** ————

If you uncheck this option, then for a processor running in HSD, the RTOS resource tabs show the state before you started that processor.

Auto Update on Timer...

Not available in this release.

RSD menu Where options are enabled, use this menu to control RSD:

Disable RSD

This option enables or disables RSD. The initial state depends on the RSD setting in the RTOS group in the board file, or .bcd file where available. See *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file* on page 15-14 for details.

If RSD is enabled, select **Disable RSD** to shutdown the Debug Agent cleanly. You can re-enable RSD after it has been disabled.

If you select **Disable RSD**, this disables all previously set RSD breakpoints. However, all HSD breakpoints are maintained.

If you select **Enable RSD**, system breakpoints are enabled but you have to enable thread and process breakpoints yourself.

Stop Target Processor

Stops the target processor, suspends RSD, and switches to HSD. Depending on your configuration settings, click the **Go** button to start execution and restart RSD.

Properties

Select this option to see details about the Debug Agent. This includes the status of the RSD module, RTOS-specific settings as defined in your board file (or .bcd file where available), RSD breakpoints, and symbols.

Note

The **RSD** menu includes the same options as the OS marker context menu in the Process Control pane. See *OS marker in the Process tab* on page 15-34 for details.

Resources toolbar

This toolbar contains three controls:

Cycle Connections button

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

This feature is not available in this release.

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

Status: This field describes the current OS module status, for example RSD.

Use the Process Control pane to see information about your system, see *Working with OS-aware images in the Process Control pane* on page 15-34 for more details.

Resources list

The Resources list is displayed in the tabbed pane at the top of the window. If you do not have RTOS support loaded, this contains only the **Conn** tab.

With an RTOS application loaded, RTOS-specific tabs are added to display the processes or threads that are configured (see Figure 15-18 on page 15-41). Click on the **thrd** tab to see the thread list available from the **Cycle Threads** button drop-down list, with the same fields shown.

In this release, the Debug Agent handles up to 64 threads and the Resources list shows all the threads on the system. This display differs from the thread list where threads that are not under the control of RealView Debugger are grayed out, see Figure 15-11 on page 15-30 for details.

Other tabs might be included to support the display of other RTOS objects, for example semaphores, memory block pools, and memory byte pools (see Figure 15-18 on page 15-41).

Details area

If you double-click on one of the entries in the Resources list, for example to specify a thread, the lower pane, the Details area, displays more information about that thread (see Figure 15-18 on page 15-41).

For more information about the meaning of the tabs and information displayed in the Resource Viewer window, see the user manual for your RTOS.

Note

Different RTOS plugins might display information in different ways in this window, for example by adding new menus. Similarly, other RealView Debugger extensions might add other tabs to the Resources list.

15.6.4 Using Action context menus

Where supported by your Debug Agent, you can perform actions on a specified RTOS resource from the Resource Viewer window. Select the required tab and then right-click on an entry to see the associated **Action** context menu.

The available actions, and the associated parameters, depend on the Debug Agent. In the example shown in Figure 15-18 on page 15-41, select the threads view and right-click on a specific thread to see the associated Action context menu options. In this case, the Debug Agent offers the possible actions:

- delete
- suspend
- resume.

Note

Interaction of RTOS resources and RealView Debugger on page 15-46 describes the interaction of RTOS actions and RealView Debugger.

The **Action** context menu also includes a **Display Details** option. This is the same as selecting **View → Display Details** for the chosen object.

If you select an action from the **Action** context menu that requires parameters, a prompt is displayed, shown in the example in Figure 15-19 on page 15-46.

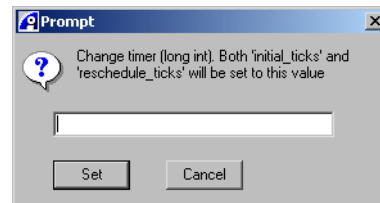


Figure 15-19 Action argument prompt

Enter the required value and then click **Set** to confirm your choice. In this example, it is necessary to update the Resource Viewer window to see the effect of this action.

The Resources list does not differentiate between captive threads and non-captive threads (see *Captive threads* on page 15-31 for details). If you try to perform an action on a non-captive thread, RealView Debugger displays an error message to say that it is not available.

See the *RealView Developer Kit v2.2 Command Line Reference* for a list of RTOS-related CLI commands you can use to carry out actions on RTOS objects. For more information about the meaning of options in the **Action** context menu, see the user manual for your RTOS.

15.6.5 Interaction of RTOS resources and RealView Debugger

Be aware of the following interactions between RTOS resources and RealView Debugger:

- The thread status, as shown in the Resource Viewer window, is fully integrated with the thread list as shown in the Code window.
- RTOS resource actions and target execution control are independent. Using actions to suspend or resume threads is independent of thread control in RealView Debugger:
 - If you suspend a specific thread, that thread is not captive in the debugger. The Resource Viewer shows the thread as suspended, but the Code window does not show that the thread has stopped, that is, it is shown as running in the Register pane or in the Call Stack pane.
 - If you resume a captive thread, the Resource Viewer shows the thread as running, but the debugger still believes that it has the thread captive. To update the debugger state, you must disconnect and then reconnect.
 - If you stop a thread from the Code window, using an action to resume does not show the thread as running, that is it still appears to be stopped.

15.7 Debugging your RTOS application

This section describes features specific to debugging multithreaded images in RealView Debugger. It contains the following sections:

- *About breakpoints*
- *Setting breakpoints* on page 15-49
- *Using the Set Address/Data Breakpoint dialog box* on page 15-51
- *Using the Break/Tracepoints pane* on page 15-53
- *Stepping threads* on page 15-54
- *Manipulating registers and variables* on page 15-56
- *Updating your debug view* on page 15-57.

15.7.1 About breakpoints

If you are in RSD mode, two types of breakpoint are available:

HSD breakpoint

This is the default breakpoint type offered by RealView Debugger. When it is hit, the breakpoint triggers and stops the processor. If you are in RSD mode when the HSD breakpoint is hit, RealView Debugger changes into HSD mode.

RSD breakpoint

Any thread that hits this breakpoint stops immediately. There are different types of RSD breakpoint:

System breakpoint

This breakpoint is set by the Debug Agent and so requires RSD to be enabled. Any thread might trigger this breakpoint. When it is triggered, a system breakpoint stops the thread that hit it, but all other threads continue.

Thread breakpoint

This breakpoint is set by the Debug Agent and so requires RSD to be enabled. A thread breakpoint is associated with a thread ID or a set of IDs, called a *break trigger group*. If any thread that is part of the break trigger group hits this breakpoint, it triggers and the thread stops. All other threads continue.

If the thread breakpoint is hit by a thread that is not part of the break trigger group, the breakpoint is not triggered and execution continues.

Process breakpoint

Not available in this release.

Using the break trigger group

The break trigger group consists of a thread ID, or a set of thread IDs, associated with a specific thread breakpoint. If any thread in the break trigger group hits the thread breakpoint, it triggers and the thread stops. All other threads, including the other threads in the break trigger group, continue.

The break trigger group is *empty* when all the threads in the group have ceased to exist. In this case, the group *disappears*. Even where the Debug Agent has the ability to communicate this information, the thread breakpoint associated with this empty break trigger group is not disabled. However, it never triggers.

If you try to reinstate a thread breakpoint where the break trigger group has disappeared, you cannot be sure that the threads specified by the group still exist or that the IDs are the same. In this release of RealView Debugger it is not possible to reinstate a thread breakpoint whose break trigger group has disappeared.

————— Note —————

A break trigger group can consist of a process, or set of processes, associated with a process breakpoint. This feature is not available in this release.

More on breakpoints

With RTOS support enabled, any breakpoint can also be a conditional breakpoint. RSD breakpoints can take the same qualifiers as HSD breakpoints and it is possible to link a counter or an expression to an RSD breakpoint.

You can also set hardware breakpoints in RSD mode but the availability of such breakpoints is determined by the debug target, that is the target processor and the Debug Agent. Hardware breakpoints are not integrated with the Debug Agent and so behave the same way in both HSD and RSD mode.

To see your support for breakpoints:

- Select **Debug → Breakpoints → Hardware → Show Break Capabilities of HW...** from the Code window main menu. This displays an information box describing the support available for your target processor.
- Select **Properties** from the **Process** tab context menu (see *OS marker in the Process tab* on page 15-34 for details). This displays an information box describing the Debug Agent support for breakpoints.

Where the memory map is disabled, RealView Debugger always sets a software breakpoint where possible. However, if you are in RSD mode and the target is running, RealView Debugger sets a system breakpoint.

Where the memory map is enabled, RealView Debugger sets a breakpoint based on the access rule for the memory at the chosen location:

- a hardware breakpoint is set for areas of no memory (NOM), Auto, read-only (ROM), or Flash.
- if the memory is write-only (WOM), or where an error is detected, RealView Debugger gives a warning and displays the Set Address/Data Break/Tracepoint dialog box for you to specify the breakpoint details.

For more details see:

- Chapter 5 *Working with Breakpoints* for a full description of HSD breakpoints in RealView Debugger.
- Chapter 6 *Memory Mapping* for a full description of memory types and access rules.

15.7.2 Setting breakpoints

When you are debugging a multithreaded image, set breakpoints in the usual way, for example:

- by right-clicking inside the **Src** or **Dsm** tab
- using the Set Address/Data Break/Tracepoint dialog box
- using context menus from the Break/Tracepoints pane
- submitting CLI commands.

To set a breakpoint in your code view:

1. Start RealView Debugger.
2. Configure RTOS support and load the multithreaded image.
3. Start the image running in RSD mode until you reach a point where threads are being rescheduled.
4. Ensure that you are working in an unattached Code window so that the current thread is visible.
5. In the **Src** tab, right-click in the gray area to the left of a source line to see the context menu.

Note

The options available on the context menu depend on your debug target and Debug Agent. Where RSD or HSD is disabled, some options are grayed out.

6. Select the required breakpoint from the list of options, that is:
 - **Set Break** sets a system breakpoint
 - **Set System Break** sets a system breakpoint
 - **Set Thread Break** sets a thread breakpoint.

The **Cmd** tab shows the breakpoint command, for example:

```
bi,rtos:thread \DEMO\#373:1 = 0x13BAC
```

Note

Process breakpoints are not available in this release.

Breakpoints are marked in the source-level and disassembly-level view at the left side of the window using color-coded icons:

- Red means that a breakpoint is active, and that it is in scope.
- Green depends on windows attachment:
 - If you are working in an attached window, green means that a breakpoint is active, but not for the thread, or process, that this window is attached to.
 - If you are working in an unattached window, green means that a breakpoint is active, but not for the current thread.
- Yellow shows a conditional breakpoint.
- White shows that a breakpoint is disabled.

If you set a thread breakpoint in this way in an unattached window, the break trigger group is the current thread. If your Code window is attached to a thread, the break trigger group consists of the thread the window is attached to.

Changing the break trigger group

If you have RSD enabled and you set a thread breakpoint, right-click on this breakpoint, marked by a thread icon, to see a context menu.

Note

The options available on the context menu depend on your debug target and Debug Agent. Where RSD or HSD is disabled, some options are grayed out.

This context menu contains options related to the break trigger group:

- Add This Thread
- Remove This Thread
- Break Trigger Group...

Note

These options are not available in this release. This means that you cannot change the threads that make up the group after the breakpoint is set from the Code window. Instead use the appropriate CLI command to modify the breakpoint (see *Using CLI commands* on page 15-58 for details).

15.7.3 Using the Set Address/Data Breakpoint dialog box

Use the Set Address/Data Breakpoint dialog box to set breakpoints:

1. Right-click in the gray area to the left of a line to see the context menu.
2. Select **Set Break...** to display the Set Address/Data Breakpoint dialog box, shown in part in Figure 15-20 on page 15-52.

Ensure that you select **Set Break...** If you select **Set Break (double click)**, the Set Address/Data Breakpoint dialog box dialog box is not displayed and a system breakpoint is set automatically.

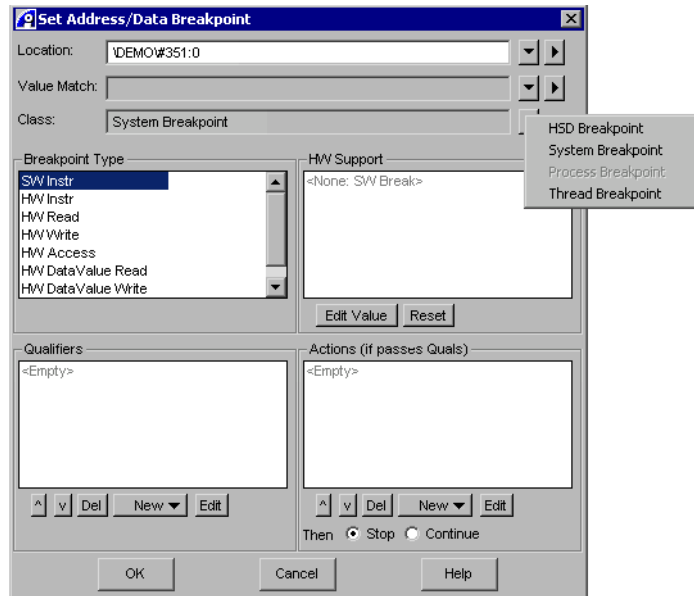


Figure 15-20 RTOS Breakpoint Class selector

Note

Chapter 5 *Working with Breakpoints* for a full description of all the controls in this dialog box.

- Use the Class field to specify the type of breakpoint to set. A system breakpoint is the default choice.

Click the drop-down arrow to the right of this field to choose from a list of available classes.

Breakpoint classes are grayed out if they are not supported by the Debug Agent.

The breakpoint Class selector options might be grayed out depending on:

- hardware capability
- Debug Agent
- HSD/RSD status, for example if RSD is not available, the field shows Standard Breakpoint
- the System_Stop setting specified in your board file

Depending on the type of breakpoint, you cannot edit an existing breakpoint where the choice is restricted by the Debug Agent. In this case, you must clear the breakpoint before you can set a new breakpoint at the same location.

15.7.4 Using the Break/Tracepoints pane

Select **View** → **Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane in the usual way, shown in Figure 15-21.

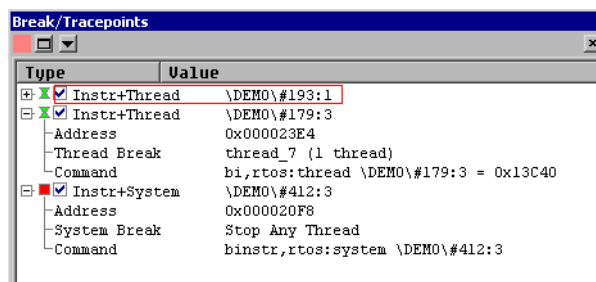


Figure 15-21 RTOS (RSD) breakpoints in the Break/Tracepoints pane

Use this pane to view the current breakpoints, or to enable, or disable, a chosen breakpoint. Breakpoints are marked using color-coded icons in the same way as in the **Src** or **Dsm** tabs (see *Setting breakpoints* on page 15-49 for details). You can also use this pane to edit breakpoints using the Set Address/Data Break/Tracepoint dialog box. Right-click on a breakpoint in the list to see the context menu where you can make changes to the breakpoint.

The Break/Tracepoints pane shows details about each breakpoint you set. What is shown depends on whether you are working in RSD or HSD mode. Figure 15-21 shows that three breakpoints are set in RSD mode. Here, two thread breakpoints are active on background threads, and a system breakpoint is active on the current thread. Figure 15-22 on page 15-54 shows three breakpoints are set in HSD mode. Here, the extra breakpoint details (available in RSD) are not included in the Break/Tracepoints pane.

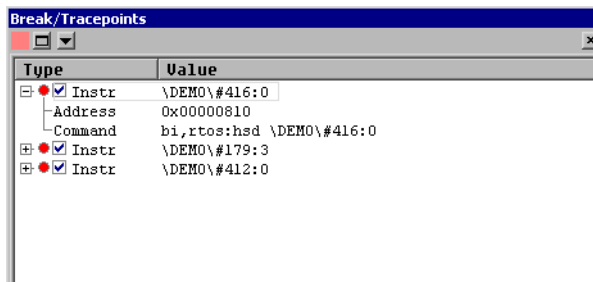
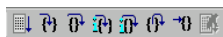


Figure 15-22 RTOS (HSD) breakpoints in the Break/Tracepoints pane

In these examples, the Break/Tracepoints pane is floating and so the pane title bar reflects the title bar of the calling Code window. Figure 15-21 on page 15-53 and Figure 15-22 show that the Code window is unattached. In this case, the Code window displays the current thread.

Chapter 5 *Working with Breakpoints* for a full description of the Break/Tracepoints pane.

15.7.5 Stepping threads



Use the Execution group from the Debug toolbar to control program execution. For example, to start and stop execution, and to step through a multithreaded application. These options are also available from the main **Debug** menu.

When you are debugging a multithreaded image, stepping behavior depends on the:

- current thread
- thread you are stepping through
- windows attachment.

Stepping in RSD mode

When you are in RSD mode, you can step any thread independently without having to stop the target. However, you must stop the thread that you want to step, for example using a system, thread, or process breakpoint.

————— Note —————

Process breakpoints are not available in this release.

If you are using an unattached Code window then you can step the current thread in the usual way. The code view changes, however, if breakpoints are hit on other background threads while you are stepping. This is because a stopped thread becomes the current thread and is visible in the unattached window.

The example shown in Figure 15-23 represents three threads at different stages of execution.

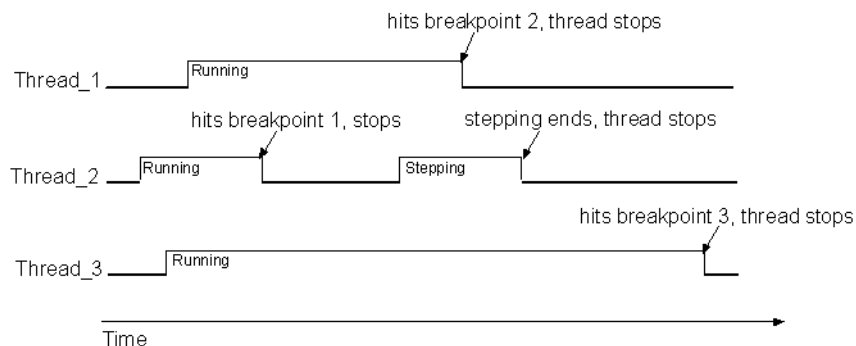


Figure 15-23 Stepping and stopping threads

In an unattached window, where Thread_2 is the current thread, the code view shows:

1. Thread_2, as stepping starts and code is examined.
2. Thread_1, when breakpoint 2 hits.
3. Thread_2, as stepping ends and the thread stops.
4. Thread_3, when breakpoint 3 hits.

In a window attached to Thread_2, the code view shows Thread_2 as stepping completes and the thread stops. In this case:

- Any stop events on Thread_1 or Thread_3 are not visible in the Code window.
- The current thread changes as breakpoints are hit. It is:
 1. Thread_2, when breakpoint 1 hits.
 2. Thread_1, when breakpoint 2 hits.
 3. Thread_2, as the debugger internal breakpoint hits when stepping ends.
 4. Thread_3, when breakpoint 3 hits.

If you want to examine a background thread you must do one of the following:

- make the background thread the current thread

- attach your Code window to the background thread
- open a new Code window and attach the window to the background thread.

Stepping in HSD mode

When you are in HSD mode, you must set a breakpoint and stop your image so that you can step through the code. When you are working on a multithreaded image, any step instruction acts on the thread that was current when the processor stopped.

15.7.6 Manipulating registers and variables

Select **View** → **Registers** to display the Register pane where you can view registers for threads in the system. If the Code window is unattached, the Register pane shows processor registers for the current thread.

If you attach a Code window to a specified thread, the Register pane displays the registers associated with the thread. These might not have the same values as the current processor registers.

You can use in-place editing to change a register value in the usual way. See Chapter 7 *Working with Debug Views* for details. However, you can only see register values, and change them, when the thread is stopped. The new register values are written to the RTOS *Task Control Block* (TCB) for the selected thread. When that thread is next scheduled, the registers used by the thread are read from the TCB into the processor.

If you are debugging ARM code, the *ARM Architecture Procedure Call Standard* (AAPCS) specifies that the first four parameters to a function are passed in registers. In addition, some local variables are optimized into registers by the compiler for parts of the function. Therefore if you modify a local variable that is stored in a register, the debugger modifies the TCB state to transfer the value into a processor register instead of modifying the target memory allocated to that variable.

———— Note ————

If you are modifying a value that you expect to be shared by several threads, for example a global variable, the compiler might have cached that value in a register for one or more of the threads. As a result, the modification you want is not propagated to all of the threads that reference the variable. To ensure that such modifications operate correctly, you must do one of the following:

- in RSD mode, modify the variable, then at the point you have stopped the relevant thread, if any thread has a cached copy of the variable, modify the copy
- in HSD mode, modify the variable, then at the point you have stopped the processor, if any thread has a cached copy of the variable, modify the copy

- in RSD or HSD, declare the variable to be volatile and recompile the program.
-

15.7.7 Updating your debug view

During your debugging session, use the Memory pane and the Watch pane to monitor execution. The Memory pane displays the contents of memory and enables you to change those contents. On first opening, the pane is empty, because no starting address has been specified. If a starting address is entered, values are updated to correspond to the current image status. The Watch pane enables you to view expressions and their current values, or to change existing watched values.

In these panes, you can use the **Pane** menu to specify how the contents are updated:

Update Window Now

If you have unselected the option **Automatic Update**, you can use this option to update the thread view manually. You can update the display using this option at any time. This enables you to catch any memory updates made externally.

Automatic Update

Updates the display automatically, that is when:

- you change memory from anywhere in RealView Debugger
- a watched value changes
- program execution stops.

This is the default.

Timed Update when Running

If you are in RSD mode, the thread view can be updated at a specified time interval during program execution. Select this option to set this timer according to the update period specified by **Timed Update Period**.

Timed Update Period

Use this to choose the interval, in seconds, between window updates.

Any value you enter here is only used when the option **Timed Update when Running** is enabled.

See Chapter 7 *Working with Debug Views* for details on working with the Memory and Watch panes, and a full description of all the options available from the **Pane** menus.

15.8 Using CLI commands

The following CLI commands are specific to OS aware debugging:

- `AOS_resource-list` (RTOS action commands)
- `DOS_resource-list` (RTOS resource commands)
- `OSCTRL`
- `THREAD`.

The following CLI commands provide options that are specific to OS aware debugging, or have a specific effect on the behavior of OS aware connections:

- `BREAKINSTRUCTION`
- `HALT`
- `RESET`
- `STOP`.

Other commands can be used with OS aware connections, such as those for stepping, accessing memory and registers, and setting hardware breakpoints.

For information on using these commands, see the chapter that describes RealView Debugger commands in the *RealView Developer Kit v2.2 Command Line Reference*.

Chapter 16

Programming Flash with RealView Debugger

This chapter describes how to use RealView® Debugger to program Flash memory on your target hardware. It contains:

- *Introduction to Flash programming with RealView Debugger* on page 16-2
- *Programming an image into Flash* on page 16-3
- *Troubleshooting* on page 16-5.

16.1 Introduction to Flash programming with RealView Debugger

RealView Debugger enables you to program Flash memory (that is, download programs or patch code and data). Unlike standard RAM-based memory, you cannot program Flash memory directly with RealView Debugger. This is because Flash has a block structure and requires various control signals to be generated to access the device.

How RealView Debugger writes to Flash memory depends on:

- the Flash type
- the specific device used
- how this device is integrated into your design.

Programming Flash with RealView Debugger requires the following files:

- Flash MMethod file
- Board/Chip Definition file.

RealView Debugger provides files for some platforms. If the files provided do not fit your requirements, then you must create your own files. How you create these files depends on whether or not your platform uses a Flash type that is already supported by RealView Debugger.

16.2 Programming an image into Flash

This section describes how to program an image to Flash. It includes:

- *Writing the image to Flash*
- *Using CLI commands to program Flash*
- *Checking the contents of Flash* on page 16-4.

16.2.1 Writing the image to Flash

To program an image into Flash using RealView Debugger:

1. Assign the BCD file that describes the memory map and specifies the FME file to the target connection.
2. Build your Flash image using the linker setting `Link_advanced.Ro_base`. Make sure that this points to a valid Flash address.
3. Connect to the target.
4. Queue your Flash image for programming into the Flash device. You can do this using one of the following methods:
 - Load the Flash image using the Load File to Target dialog box, or select an image from the **Target** → **Recent Images** list. See Chapter 3 *Working with Images* for instructions on how to load an image.
 - Load the Flash image using the Upload/Download file from/to Memory dialog box.

In both cases, the Flash Memory Control dialog box is displayed.

5. Click **Write** on the Flash Memory Control dialog box to write the image to the Flash device. RealView Debugger uses the information and Flash routines in the FME file that is specified in the BCD file.

See Chapter 8 *Reading and Writing Memory, Registers and Flash* for instructions on how to use the following dialog boxes:

- Upload/Download file from/to Memory dialog box
- Flash Memory Control dialog box.

16.2.2 Using CLI commands to program Flash

You can use the following CLI commands to program an image to Flash:

- `LOAD`, to load the Flash image, for example:
`LOAD 'C:\my_projects\flash\my_flash_image.axf'`

- FLASH, to perform operations on Flash. For example, to write the image to Flash enter:
FLASH,write

For more details on how to use these commands, see *RealView Developer Kit v2.2 Command Line Reference*.

16.2.3 Checking the contents of Flash

You can check the contents of Flash as follows:

1. Make sure that the Memory pane is visible. If it is not, then select **View** → **Memory** from the Code window main menu.
2. In the Memory pane, right-click on an address in the left-hand column.
3. Select **Set Start Address...** from the context menu.
4. Enter the start address of your Flash memory.
 - for the Evaluator-7T example, enter 0x1800000
 - for the Integrator/AP, enter 0x24000000.

The contents of memory starting at this address are displayed in the Memory pane. The contents of Flash are shown in green.

For more details on using the Memory pane, see Chapter 8 *Reading and Writing Memory, Registers and Flash*.

16.3 Troubleshooting

If you are having problems programming an image to Flash, check the following:

- Does the BCD file correctly describe the Flash memory area for your board, and does it specify the correct FME file?
- Does your target connection have the correct BCD file assigned to it?
- Is the icon for the Flash area in the **Map** tab green, and are the Flash memory locations in the Memory pane shown with a yellow background?

This indicates that the Flash_type setting does not point to a valid FME file, or is not set.

- Is the image you are loading a valid Flash image?

Use the `displflash` utility to display information about the FME file, and examine the following line:

Routine Code PC-rel. Can load at *address*

The *address* must be a valid RAM area on your board. If this is not, then:

1. Display the Project Properties for the project that you are using to build the Flash image.
2. Change the value for the linker setting `BUILD.Link_advanced.Ro_base` to a valid value.
3. Rebuild the FME file.

————— **Note** —————

If you built the image using the compilation tools directly, see the `--ro_base` linker option in the *RealView Developer Kit v2.2 Linker and Utilities Guide*.

- Does your Flash image have the same endianness as the FME file?
 - Build your Flash image and FME file with the correct endianness. Set either the `Compilation.Endianness` setting for the ARM compiler or `Assembly.Endianness` setting for the ARM assembler in the Project Properties.
To check the endianness of the FME file, use the `displflash` command.
 - If you built the image using the compilation tools directly, see the `--littleend` or `--bigend` options in *RealView Developer Kit v2.2 Compiler and Libraries Guide* or in *RealView Developer Kit v2.2 Assembler Guide*.
 - Make sure that your target hardware and connection configurations have the correct endianness assigned.

- If you have created your own FME file:
 - Have you correctly specified the Flash and board settings in the AME files?
 - Are your Flash algorithms correct?

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-8.

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.

DEBUGGER contains two second-level groups and a file:

- *Command*
- *Disassembler* on page A-3
- *Board_file* on page A-5.

A.1.1 Command

Settings in this group control the behavior and appearance of the Code window command line and Output pane. Use these to customize the input and output format used in this area.

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 pane. 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 <code>ce number=0xABCD</code> . Switching to hex also enables you to enter decimal numbers, for example <code>ce number=01234t</code> .
Radix_out	This setting specifies number format output options at start-up. The default format is decimal.

Table A-1 Command settings

Name	Properties
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 scrollbar 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.

Table A-2 Disassembler settings

Name	Properties
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 Src tab in the File Editor pane.
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.
Register_syms	Not available with ARM® tools. Some disassemblers identify register usage in the operand fields. Use this to show the corresponding register-based variable when possible.
Format	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. You can also change the format dynamically using the DISASSEMBLE command, that is DISASSEMBLE /D, for default format, or DISASSEMBLE /A, for alternate format. When debugging ARM processors, use this to force the disassembler to display 16-bit values, as opposed to showing the appropriate format for the given processor context, that is mixed 16-bit and 32-bit values.
Instr_value	In general, disassemblers show instructions as values and as opcodes or operands. Use this to suppress the value display where possible.

A.1.3 Board_file

Change this setting to specify a different board file for the current session.

Changing this value takes immediate effect. The specified board file is read and the contents used to populate the configuration details.

Resetting the value back to Empty does not take effect until the next time RealView Debugger starts.

A.2 CODE

Settings in this group govern the behavior of all Code windows when running a debugging session. These settings control the display characteristics of windows, their size and position, and any user-defined buttons created on the toolbars (not available in this release).

CODE contains two second-level groups and a settings rule:

- *Pos_size*
- *Button*
- *Asm_type* on page A-7.

A.2.1 Pos_size

Settings in this group control the position on screen and the size of Code windows in lines and characters. Use these to customize the size and position of Code windows in the debugger.

On start-up, RealView Debugger uses the last-used settings unless overridden by these settings. As you open new Code windows in the session, they are controlled by these settings.

Table A-3 describes the settings.

Table A-3 Pos_size settings

Name	Properties
Num_lines	Use this to specify window height as a given number of lines. The default is 0x0200.
Num_chars	Use this to specify window width as a given number of characters. The default is 0x02A0.
X_pos	Use this to specify the X position of the top-left corner of the window. The default is 0x010F.
Y_pos	Use this to specify the Y position of the top-left corner of the window. The default is 0x0066.

A.2.2 Button

———— **Note** ————

These settings are not available in this release.

A.2.3 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
- there are no user-defined projects currently open
- there are no auto-projects currently open.

In this case, RealView Debugger does not know the format of instructions and so cannot define source coloring rules. This generates a selection box where you can specify the processor type.

Change this to specify a default processor type on start-up, for example ARM.

Saving a change to this setting takes immediate effect on new assembler source files opened following the update.

A.3 ALL

This group contains three second-level groups:

- *Text*
- *Search* on page A-11
- *Edit* on page A-12.

A.3.1 Text

Settings in this group control the File Editor pane 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 File Editor pane.

Width Use this setting to specify the width, in number of characters, for text displayed in the File Editor pane.

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-4 describes these settings.

Table A-4 Internationalization settings

Name	Properties
Enabled	Use this to enable or disable internationalization. By default, internationalization is disabled, that is this setting is False.
Language	Use this to specify the language to use for text. The options are: <ul style="list-style-type: none"> English (default) Japanese Chinese.
Default_encoding	Use this to specify the default text encoding. The options are: <ul style="list-style-type: none"> ASCII (default) UTF-8 Locale.

Font_information

This group contains the Pane_font setting to set the font used in panes. A Font dialog box is displayed to enable you to change the font settings shown in Table A-5. This table also shows the default font settings.

Table A-5 Default font settings

Setting	Default Value
Font	Lucida Console
Font style	Regular
Size	10
Script	Western

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-6 describes the settings.

Table A-6 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	<p>The color scheme you want to use for source coloring. You can select one of:</p> <ul style="list-style-type: none"> • Default for the RealView Debugger v1.8 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 File Editor pane.
Numbers_bkgrnd	Use this to specify the background color for numbers displayed in the File Editor pane.
Strings_text	Use this to specify the foreground color for strings displayed in the File Editor pane.
Strings_bkgrnd	Use this to specify the background color for strings displayed in the File Editor pane.
Keywords_text	Use this to specify the foreground color for C/C++ keywords displayed in the File Editor pane.
Keywords_bkgrnd	Use this to specify the background color for C/C++ keywords displayed in the File Editor pane.
Comments_text	Use this to specify the foreground color for comments displayed in the File Editor pane.
Comments_bkgrnd	Use this to specify the background color for comments displayed in the File Editor pane.
Identifiers_text	Use this to specify the foreground color for identifiers displayed in the File Editor pane.
Identifiers_bkgrnd	Use this to specify the background color for identifiers displayed in the File Editor pane.
User_text	Use this to specify the foreground color for user-defined keywords displayed in the File Editor pane.

Table A-6 Source_coloring settings

Name	Properties
User_bkgrnd	Use this to specify the background color for user-defined keywords displayed in the File Editor pane.
Preprocessor_text	Use this to specify the foreground color for preprocessor keywords (#keyword) displayed in the File Editor pane.
Preprocessor_bkgrnd	Use this to specify the background color for preprocessor keywords (#keyword) displayed in the File Editor pane.
Operator_text	Use this to specify the foreground color for operators displayed in the File Editor pane.
Operator_bkgrnd	Use this to specify the background color for operators displayed in the File Editor pane.
User_keywords	Use this to specify a list of user-defined keywords that are highlighted when they appear in the File Editor pane.

A.3.2 Search

Settings in this group control the searching behavior when working with source files in the File Editor pane.

These settings can be overridden dynamically using the menus and toggles in the File Editor.

Table A-7 describes these settings.

Table A-7 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.

Table A-7 Search settings

Name	Properties
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 File Editor pane.

These settings can be overridden dynamically using the menus and toggles in the File Editor.

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 File Editor pane.

Table A-8 describes these settings.

Table A-8 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 pathname to a new location, for example to keep all backup files in one special directory.
Backup_ext	By default, backup files are saved with the .bak extension appended to the original filename.

Tab_conv

Settings in this group control the display behavior when working with source files in the File Editor pane. 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-9 describes these settings.

Table A-9 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

Settings in this group control source control access tools when working in the File Editor pane and in the debugger. The editor attempts to detect any source control system in use, where possible, but you might have to specify it before RealView Debugger can use it.

Use these settings to specify complete source control commands, and to override commands for known systems.

The Src_ctrl group contains a low-level group and a series of settings:

Cmnds Use these to specify source control commands for use with RealView Debugger.

Src_ctrl settings

Use these to specify the source control system to be used with RealView Debugger to control access to files.

For full details on these settings see the chapter that describes version control in *RealView Developer Kit v2.2 Project Management User Guide*.

Edit

Settings in this group configure editor behavior when working with source files in the File Editor pane.

Table A-10 describes these settings.

Table A-10 Edit settings

Name	Properties
Drag_drop_dis	Use this to disable drag-and-drop editing when working in the File Editor pane.
Vi	Not used.
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 disabled in the debugger and the File Editor. Use this to change the editor default to show 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-10 Edit settings

Name	Properties
Startup	<p>The default start-up file, that is <code>rvdebug.sav</code> 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.</p> <p>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.</p>
Template	<p>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.</p> <p>By default, the file is named <code>rvdebug.tpl</code> and is saved in your home directory, or in the default settings directory <code>\etc</code>. Use this to change this pathname.</p>
Restore_state	Not used.

Appendix B

Configuration Properties Reference

This appendix contains reference details about board file entries that define target configurations and custom connections. It contains the following sections:

- *About this appendix* on page B-2
- *Target configuration and connections* on page B-3
- *Generic groups and settings* on page B-5
- *Target configuration reference* on page B-8
- *Custom connection reference* on page B-26.

B.1 About this appendix

You configure the way that RealView® Debugger connects to, and interacts with, your debug target using a board file. The default board file, called `rvdebug.brd`, is set up when you first install RealView Debugger.

The contents of the Connection Control window are defined by elements of the board file. To change target configuration, or to add new connections, you change the entries in the board file and this modifies the elements displayed in the Connection Control window. RealView Debugger provides a GUI interface, the Connection Properties window, to make these changes. See *Connection Properties window* on page 12-5 for details.

To follow this appendix, you must display the Connection Properties window, either:

- Select **Target → Connection Properties...**
- Select an entry in the Connection Control window. Right-click and select **Connection Properties...** from the context menu.

Use this method to open the Connection Control window at the parent group for the chosen connection, target or processor.

This appendix describes the settings and groups of settings that appear in the board file. It assumes that you are using the RealView Debugger base product that includes built-in configuration files to enable you to make a connection. If you have changed these files, or created new target configuration files of your own, your window looks different.

B.2 Target configuration and connections

RealView Debugger makes a distinction between target configuration, and how a target is accessed, target connection. Both are configured using different types of entry in the board file.

B.2.1 Types of entry

A CONNECTION group is normally used to specify connection details, how the target is accessed. This includes the nature and addresses of the hardware interface, for example the port name of the JTAG interface that is connected to the target. This information is described in the Connect_with block of the board file and in the file associated with the Configuration setting.

Note

Occasionally, you can use a DEVICE group in place of a CONNECTION group.

Within a specific CONNECTION, one or more BoardChip_name entries are used to associate the connection with a BOARD, CHIP, or COMPONENT description that defines peripherals and memory maps.

It is recommended that the descriptions of the target are only defined in BOARD or CHIP definitions, and that these descriptions are stored in .bcd files. For example, the definition of the registers and peripherals of the ARM® Integrator™/AP motherboard is stored in the file AP.bcd. Figure B-1 shows how these groups are linked.

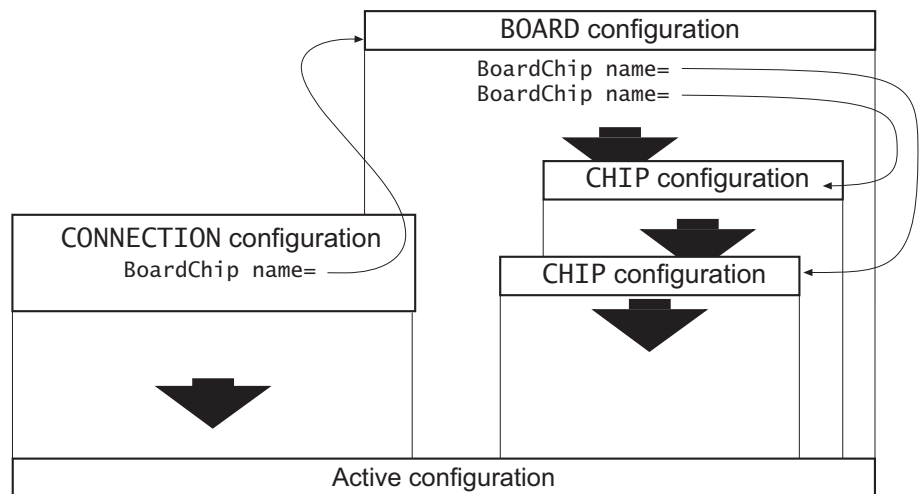


Figure B-1 How connections, boards, and chips fit together

Therefore, the board file consists of the following types of entry:

- Connection entries using the following groups:
 - CONNECTION or DEVICE
 - RVBROKER.
- Configuration entries using the following groups:
 - BOARD
 - CHIP
 - COMPONENT.

This appendix describes the groups and individual settings that appear in the board file. It assumes that you are familiar with the contents of the Connection Properties window as described in *Connection Properties window* on page 12-5.

B.3 Generic groups and settings

There are several board file entries that are common to many of the settings groups, shown in the example CONNECTION=RVI-ME entry in Figure B-2.

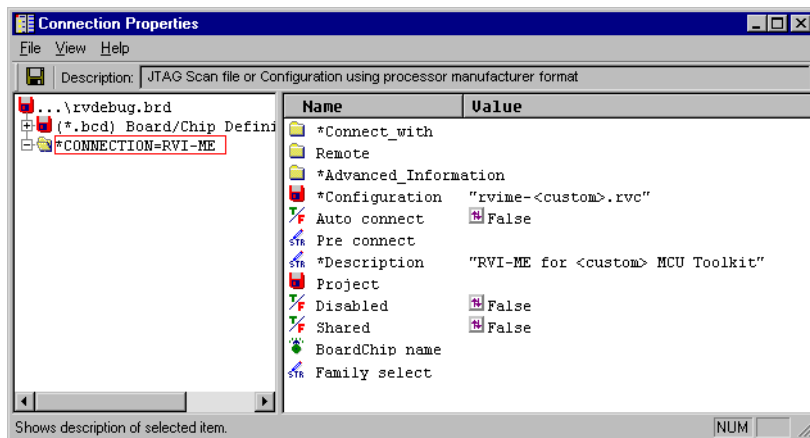


Figure B-2 Viewing generic settings

These settings are found in CONNECTION groups. In some cases, some or all of these settings are also found in other groups, for example BOARD=AP in the AP.bcd entry. This section describes these generic groups and settings:

Connect_with	Specifies the vehicle making the connection. It includes the settings values:
Manufacturer	The name and the type of the connection. Right-click to see a list of available connection types, but you also require an appropriate license and the hardware to use them.
IOdevice	This field contains additional information about the target hardware. Not available in the current release of RealView Debugger.
Speed	You can set the emulation speed for some emulators. Not available in the current release of RealView Debugger.

Remote	This specifies how remote connections are configured. It includes the settings values:
Hostname	The hostname, or IP address, of the remote workstation.
Port	Specifies that a non-default TCP/IP port is used to make the connection to the remote workstation.
Advanced_Information	<p>Provides ETV target configuration information about the debug target.</p> <p>This feature is described in detail in <i>The Advanced_Information block</i> on page B-8.</p>
Configuration	<p>Specifies a named configuration file.</p> <p>The default is to search for the .jtg file with the same name as this group (after the = sign). The name in this field does not have to be the same as the name of this group. If the file specified here is a full pathname, then only that location is used. Otherwise, it is searched for.</p> <p>When working with RVI/RVI-ME targets, .jtg files are replaced by .rvc configuration files. When working with custom simulator connections, .jtg files are replaced by .auc configuration files.</p>
Aux_Configuration	This setting has no purpose in this release.
Auto_connect	This setting has no purpose in this release.
Pre_connect	Forces an order for device connection. When you connect to a device within the .jtg file, this ensures that one or more specific devices are connected first regardless of which device is selected for connection. This enables pre-setup of the specific devices to guarantee correct operation, such as initializations. You can specify the device(s) to connect to first by name, by processor name, or by processor type, as used in .cnf files that contain target configuration settings.
Description	Description of what board, processor, or emulator this is for.

Project	<p>Opens one or more projects automatically when connecting to this board. If more than one device is in the scan chain and they are the same processor type, you must set the <code>Specific_device</code> field of the project to bind the project to the correct device(s). If the devices are different processor types, this is not necessary.</p> <p>See <i>RealView Developer Kit v2.2 Project Management User Guide</i> for details on working with projects in RealView Debugger.</p>						
Disabled	Disables this entry so that it is grayed out in the Connection Control window.						
Shared	Enables the sharing of target configurations for remote connections.						
BoardChip_name	Refers to the BOARD, CHIP, or COMPONENT group this connection is derived from, using its name. If the group has more than one name separated by a slash, such as ID/name, any of them can be used. If not specified, the name of this group is used to match a board or chip.						
Family_select	<p>Ensures the correct family member is used (for example, for memory mapping, and registers) when the silicon ID is ambiguous. Some chip families do not use different silicon IDs for different members of the family, and this field enables you to specify which you are using. Specify the family member using one of these formats:</p> <table> <tr> <td><code>name=family_name</code></td><td>This enables you to specify the name of the device. Use either the name defined in the .jtg file or the processor name. This is used when multiple chips are housed on the same target but from different families.</td></tr> <tr> <td><code>family_name</code></td><td>Choose from the preconfigured list.</td></tr> <tr> <td><code>silicon_id</code></td><td>Can be expressed as <code>num.num.num.num</code>, for example 15.255.15.15, or as a value, for example 41029401.</td></tr> </table>	<code>name=family_name</code>	This enables you to specify the name of the device. Use either the name defined in the .jtg file or the processor name. This is used when multiple chips are housed on the same target but from different families.	<code>family_name</code>	Choose from the preconfigured list.	<code>silicon_id</code>	Can be expressed as <code>num.num.num.num</code> , for example 15.255.15.15, or as a value, for example 41029401.
<code>name=family_name</code>	This enables you to specify the name of the device. Use either the name defined in the .jtg file or the processor name. This is used when multiple chips are housed on the same target but from different families.						
<code>family_name</code>	Choose from the preconfigured list.						
<code>silicon_id</code>	Can be expressed as <code>num.num.num.num</code> , for example 15.255.15.15, or as a value, for example 41029401.						

For information about connection configuration, including the CONNECTION and DEVICE entries, see *Custom connection reference* on page B-26.

B.4 Target configuration reference

This section describes in detail the target configuration entries supported by RealView Debugger. It contains the following sections:

- *The Advanced_Information block*
- *BOARD, CHIP, and COMPONENT groups on page B-24.*

B.4.1 The Advanced_Information block

The Advanced_Information block enables you to provide ETV information about the debug target, for example extended memory mapping, mapped registers, and peripherals. Because more than one device might be present, and each might have different details, you can create more than one Advanced_Information block. The base group is called Default and is used if you provide no other information. Entry names can be the same as processor or device names in configuration files, for example in the .rvc file, or .jtg file, for the connection. These apply more specifically to matching devices.

Advanced information settings can be nested so that one might refer to another, which in turn might refer to another. These references cause the information to be concatenated. References are made to board and chip definitions.

The Default group in an Advanced_Information block contains:

- *Application_Load* on page B-9
- *Memory_block* on page B-9
- *Map_rule* on page B-13
- *Register_enum* on page B-14
- *Register* on page B-14
- *Concat_Register* on page B-15
- *Peripherals* on page B-16
- *Register_Window* on page B-17
- *ARM_config* on page B-18
- *Cross_trigger* on page B-20
- *RTOS* on page B-20
- *Pre_connect* on page B-21
- *Commands* on page B-21
- *Connect_mode* on page B-22
- *Disconnect_mode* on page B-23
- *Id_chip* on page B-24
- *Id_match* on page B-24
- *Chip_name* on page B-24

- *Endianness* on page B-24.

It is recommended that you copy the Default group to create your own, named, groups. Do not delete the Default group so that it is available for future configuration tasks.

Application_Load

Use the Application_Load group to change the way that an executable image is loaded into target memory. The default is to write the memory using the emulator or EVM board. Settings in this group can be used to override the default and so disable all image load, for pure ROM or EPROM systems, or to run an external program to do the load.

The Application_Load group contains the following settings:

Load_using Specifies how to perform the load.

Load_command

This defines a shell command run to perform the load. The command might contain \$ variables which are substituted by RealView Debugger before calling. The possible \$ variables are:

\$D	directory of the application
\$P	full path of the application
\$F	filename of the application
\$N	name of the application without the extension.

If the command starts with an exclamation mark (!), the return value of the shell command is not used to stop the load, otherwise a non-0 return aborts the load. In all cases, the output of the command is shown in the **Log** tab in the Output pane.

Load_set_pc This controls how the PC is initialized during an image load. The default is to set the PC to the entry point if an entry point is defined and symbols are loaded and this is not an appended load (it is replace or new). This enables you to disable setting the PC under any situation, or to set it specifically to address 0.

Memory_block

Entries in the Memory_block group are used to build up fixed, enabled, or based memory regions. A fixed block is one that refers to memory that is always enabled, always at the same place, and always the same size. An enabled memory block is enabled or disabled by a register value (see *Map_rule* on page B-13). A based memory block has its start or length adjusted or set by a register value. This value might be added to an offset or might itself be the required value.

This contains the following settings:

Attributes Additional attributes can be specified for the memory. These are used by simulator models to guide the debugger when accessing this block of memory. The following settings are available:

Internal This memory is internal to the processor core and not treated as external. This affects wait-state timings and other factors.

Access-rule

The access rule information is only used by simulators to control timing issues. It is noted if a link command file is generated.

Access_size

Use this field to control how RealView Debugger accesses memory internally.

By default, this is set to 0, to indicate the default memory access size for the processor, for example, byte-size for ARM processors.

Specify a value greater than 0 to set access size in bytes, for example 1 or 2. If you do this, you must also specify a size for Memory_block, otherwise RealView Debugger displays an error message.

For external memory with only byte-wide or halfword-wide access enabled, this can be used to ensure proper access to the memory. Depending on the processor, this might have no effect.

Volatile Set to True if access destroys contents or the contents change in response to external events.

Shared This field indicates if the memory is shared with other processors. If it is, it also indicates if directly shared (accessed directly using the bus) or indirectly shared (using the host workstation port of this processor). If this is set, this field indicates what other processors or devices see this memory.

Shared_id

This field contains a number that identifies this memory block. You must use the same number for each device when referring to it. RealView Debugger can then correctly update all device views when this memory is modified.

Register_Pos_Len

Used when one or two memory-mapped registers are used to set the base address and length of the memory block, such as for cross-bar switches, and chip-selects. These are not used for enables which are set using map rules (see *Map_rule* on page B-13). The following settings are available:

Register_base

Enables you to specify a memory block position based on a memory-mapped register. The value of the register is added to the start field to construct the block start address. It can be masked and scaled (multiplied or divided) first.

Base_mask

Mask to apply to register.

Base_scale

Use this to change the value of the register contents, after masking, to define the actual base. If the number is positive, it is multiplied against the register content. If the number is negative, it is divided from the register content.

For example, a byte register might select the 64Kbyte region to map to. If the scale is $0x10000$ (64K), then register values of 0, 1, 2, 3, ... each select a 64Kbyte region. If the selector occupies part of a register, the mask is applied to select only the selector portion and the scaling value itself might be scaled. Using this example, if the byte selector portion is the upper byte of a register, the scale value is $0x10000/0x100=0x100$ (256). So, mask with $0xFF00$ and multiply by 256 to get a 64Kbyte selection.

Register_length

Enables a block of memory to be sized by a register. The content of the register is added to the specified length to compute the block length.

Len_mask

Mask to apply to register.

Len_scale

Used like *Base_scale*.

Len_table

Enables table indexing for the length. The length register is masked and scaled and then used as an index in a table of values. The last value is used if the scaled register value is too large. The table value is added to the length field of the block.

Update_rule

Indicates how often to check the register to see if the mapping has changed. For cases where the mapping is set by jumpers, which read as registers, it must be inspected only when first connecting to the device. If the program changes it, it must be tested on each stop. Valid values are:

<code>init_time</code>	Test when connecting to device.
<code>update_init</code>	Test on connect and when register changes.
<code>stop_update</code>	Test on connect, change, and execution stop.

Start	The base address of the block. If the block is mapped using a register, this is the offset from that register.
Length	The block length of this memory unit. If Length is set by a register, this must be 0 or the amount to add to that register.
Type	<p>The type of memory is dependent on the device type. The default is to map to data space. Otherwise, a memory space can be specified. Valid values are:</p> <ul style="list-style-type: none"> • <code>default</code> • <code>program</code> • <code>data</code> • <code>I0</code>.
Access	<p>Indicates how the memory is to be treated. For simulators, this affects the target use of the memory. For hardware targets, this only affects how the debugger uses the memory and any generated linker command files.</p> <p>Right-click on this setting to see available access types.</p>
Wait_states	Used with simulators to calculate the cycles used when accessing this memory. The default is based on the wait-state model used by the processor for external memory. This value is noted when link command files are generated from this data to enable careful positioning of sections to this memory.
Flash_type	<p>Contains the name of a file containing the Flash programming code and information for this processor. These files have the file extension <code>.fme</code>.</p> <p>By using the routines in this file, RealView Debugger can erase, modify and verify the contents of Flash memory.</p>

For instructions on how to create your own FME file, see Chapter 16 *Programming Flash with RealView Debugger*. For more details on how to use the FME file, see Chapter 8 *Reading and Writing Memory, Registers and Flash*.

Description Description of memory space.

Volatile Enables you to define a memory block that is volatile on read (and so is marked specially in the Memory pane). The format is an offset from within this block (0 relative). A range can be specified, for example 0x10..0x20 or 0x40..+4. If not a range, it defines a single value.

Map_rule

These entries control the enabling and disabling of memory blocks using target registers. You specify a register to be watched, and when the contents match a given value, a set of memory blocks is enabled. You can define several map rules, one for each of several memory blocks. The following settings are available:

Register This is the name of a memory-mapped target register that controls the visibility of a memory block. This register is read to determine the current mappings. You must define the register using the Register block (see *Register* on page B-14).

You are recommended to name the register itself instead of the bit fields within it when more than one bit field controls the mapping.

Mask This is ANDed with the contents of the register as described in Value.

Value This is compared with the register contents after the mask is added. The comparison is (reg-value & mask) == value. For example, if bit 3 being HIGH means the map is enabled, mask is 0x8 (1<<3) and value is also 0x8.

On_equal This contains the name of one or more memory blocks to enable when the value matches, or disable when it does not match. To replace one block with another, create one rule that tests for one value and another that tests for a different value.

Update_rule

Indicates how often to check the register to see if the mapping has changed. For cases where the mapping is set by jumpers, which read as registers, it must be inspected only when first connecting to the device. If the program changes it, it must be tested on each stop. Valid values are:

init_time Test when connecting to device.

update_init Test on connect and when register changes.

stop_update Test on connect, change, and execution stop.

Register_enum

Enumerations can be used, instead of values, when a register is displayed in the Register pane. This setting enables you to define the names associated with different values. Names defined in this group are displayed in the Register pane, and can be used to switch the register.

Register bit fields are numbered 0, 1, 2,... regardless of their position in the register.

The following setting is available:

Names You can specify a list of names, either in the form *name,name,name,...* or in the form *name=value,name=value,name=value,...*

Register

This group enables you to define memory-mapped registers provided at the board or ASIC level. Each register is named and typed and can be subdivided into bit fields (any number of bits) which act as subregisters.

The Register group contains a Default group which contains a child group called Bit_fields which, in turn, contains the Default settings:

Position Bit position from 0 (LSbit).

Size Size in bits.

Signed Set to True if signed, otherwise set to False.

Enum Enumeration name to show values in the Register pane, derived from the Register_enum group.

Read_only Set to True if read-only (cannot modify).

Volatile Indicates that the register has side effects when it is read or written. A common read side effect is loss of data (when pulled from a UART, for example). A common write side effect is for the device to take some action on write (triggering a DMA, for example). This information is used in the Register pane. Right-click to see available options.

Gui_name Optional name for showing in Register pane.

Other settings in the Register entry, Default group, are:

Start The base address of the block. If the register is mapped using a memory block, this is the offset from that register (see Base).

Length	The block length of this memory unit. If Length is set by a register, this must be 0 or the amount to add to that register.
Base	This specifies how to interpret Start. If Base is Absolute, Start is the memory address of the register. Otherwise, Base can be set to the name of a memory block, and Start is an offset from the base address of that memory block.

Memory_type

The type of memory is dependent on the device type. The default is to map to data space. Otherwise, a memory space can be specified. Valid values are:

- default
- program
- data
- IO.

Type	Specifies how to interpret the value contained in the register. The type names are as in the C language.
Read_only	If set to True, the register is read-only and the debugger does not let you write to it. Otherwise, you can modify the value using the Register pane in the Code window and using CLI expressions.
Write_only	If set to True, the register cannot be read. The debugger does not attempt to query the hardware for the current value when the Register pane is displayed.
Volatile	If set to True, the register value can change without the debugger explicitly modifying it. For example, a hardware timer continues to count even when the processor is halted.
Enum	The name of a Register_enum block that maps a register value to a textual string describing the value.
Gui_name	The name of the register as it appears in the Register pane.

Concat_Register

You can define a concatenated register that is built up using specific bits from other registers. Concatenated registers are usually used only for memory mapping, but you can also use them for control and status. The suggested approach is to name two registers and then shift and mask them into the new register. If you want to concatenate parts from more than two registers, you can build them up in stages.

The following settings are available:

Low_name	Name of low register.
Low_shift	Amount to shift low register by (<0 for left shift).
Low_mask	Amount of low register to mask (after shift).
High_name	Name of high register.
High_shift	Amount to shift high register by (<0 for left shift).
High_mask	Amount of high register to mask (after shift).
Length	Length of register, in memory units.
Type	An explicit type for the register. If you do not specify the type, the default is the signed scalar C type based on the register size.
Enum	Enumeration name to show values to show values in the Register pane, derived from Register_enum group.
Gui_name	Optional name for showing in Register pane.

Peripherals

This group enables you to define block peripherals so they can be mapped in memory, for display and control, and accessed for block data, when available. You define the peripheral in terms of the area of memory it occupies (for all its registers), and a breakdown of the registers used for access and control. The following settings are available:

Access_Method

This applies only when you can access blocks of data, and contains:

Type Method used to extract data.

Method_name

Name of access method function if required.

Start Buffer or DMA start address.

Length Buffer or DMA length.

Register Used to add memory mapped registers provided at the board or ASIC level. Each register is named and typed and can be subdivided into bit fields (any number of bits) which act as subregisters. See *Register* on page B-14 for details.

Start Start address of first peripheral register.

Length	Block length.
Base	Controls how the start field is interpreted. The default is Absolute (from 0), but can be relative to a memory block (if the block is disabled, the peripheral is too).
Type	Basic type of the device. The available values are: <ul style="list-style-type: none"> • serial • parallel • block • network • display • other.

Description Description of the device.

Register_Window

This entry contains a set of lines to show in the Register pane. The name of the block is the tab name used for the lines. Each line contains a list of mapped registers displayed in the Register pane, see *Register_enum* on page B-14, *Register* on page B-14, and *Concat_Register* on page B-15 for more details.

The format of a line is *name,name,name,...* where each *name* is the name of a register or bit field. Be aware of the following:

- If the string starts with an equals sign, =, all the registers are shown as name=value in the window, else shown in table form (the name is above the value).
- If a line starts with an underscore character, _, the line shows as a comment label (non-active).
- If the line starts with an exclamation mark, !, it provides a description line for the tab.
- If the line starts with:
 - \$ the next line starts or ends an expansion block, controlled by + or -.
 - \$+ indicates a collapsed block
 - \$- indicates a expanded block
 - \$\$ this ends a previously opened block.

ARM_config

This group enables control of ARM processor settings used for ARM emulators, monitors, or simulators. These control features such as semihosting, vector catching, and memory control (for stack and heap assignment) which must be set or unset depending on the type of runtime you have linked into your application.

You can also set many of these at runtime using pseudo-registers. To do this, name the block Default if it applies to all devices or give it the name of the scan chain device to which it applies.

The following settings are available:

Stack_Heap The ARM tools automatically set the stack and heap based on the top_of_memory using semihosting. The following settings are available:

Stack_bottom	Bottom of stack (lowest address).
Stack_size	Size of stack in bytes.
Heap_base	Bottom of heap (lowest address).
Heap_size	Size of heap in bytes.

Vectors If Vector_catch is set to True, the fields within this group enable individual control over each vector.

The following settings are available:

Reset	Set this to catch Reset exceptions.
Undefined	Set this to catch undefined instruction exceptions.
SWI	Set this to catch Software Interrupts.
P_Abort	Set this to catch Prefetch abort exceptions.
D_Abort	Set this to catch Data abort exceptions.
Address	Set this to catch Address exceptions.
IRQ	Set this to catch normal interrupt exceptions.
FIQ	Set this to catch Fast Interrupt exceptions.
Error	Set this to catch errors.

Semihosting Enables programs to communicate with the host workstation. Semihosting operations supported include stack and heap assignment and console I/O (printf and scanf type calls). Semihosting is implemented using the SWI instruction. You can change the semihosting vector during debug using the @semihost_vector pseudo-register.

On some targets you can also define a window or file number to display semihosting printf messages using setreg @SEMIHOST_WINDOW=*number*. The window numbers match the VOPEN command numbers, and the file numbers match the FOPEN command numbers. However, if your program displays prompts, it is suggested that you use a window.

Contains:

Enabled	Set this to enable semihosting.
Vector	Address of SWI vector catch to use.
Arm_swi_num	ARM SWI instruction for semihosting.
Thumb_swi_num	Thumb® SWI instruction for semihosting.

Armulator Contains:

Clock_speed	Clock speed in MHz as num.num.
Fpoint_emu	True if floating point emulation.
Config_file	The name of the configuration file.

Top_memory

Enables the semihosting mechanism to return the top of stack and base of heap. If not defined here, the default for RVI/RVI-ME is used. Any defined value is set into each tool to force this address base. You can use explicit stack and heap sizes and locations below, but this might not be supported by all debug targets. You can also set this during a debugging session using the @top_mem pseudo-register.

Vector_catch

Used to catch possible program errors by setting breakpoints on (or otherwise trapping) the vectors. The default is to catch error-type vectors but leave IRQ, FIQ and SWI alone. SWI is caught separately by semihosting if enabled. To use this, the vectors must be writable. These can also be set during debugging using the @vector_catch pseudo-register. In this case, each bit, starting with 1, represents the vectors from reset to FIQ.

Properties

This enables free-form definition of the properties required by a vehicle (emulator or simulator). The form of the string is *name=value*, where *name* is the name for the property as defined by the vehicle and *value* is a numeric value in hex or decimal.

Cross_trigger

These settings control the cross-triggering of a stop command between multiple processors that are closely coupled in hardware. They specify whether stopping execution of one processor stops execution of other processors, because of a break or other stop condition:

- input triggering means that the processor is stopped by others
- output triggering means that the processor can stop others.

The available settings are:

Trig_in_ena	List of commands to enable input triggering.
Trig_in_dis	List of commands to disable input triggering.
Trig_out_ena	List of commands to enable output triggering.
Trig_out_dis	List of commands to disable output triggering.

RTOS

This group enables automatic loading of RTOS and kernel awareness. This group enables you to configure an RTOS-enabled connection when you have no vendor-supplied .bcd file. When the RTOS is symbol-hooked, the RTOS plugin is only loaded when the RTOS, or its symbols, are loaded to the target. See the instructions from your vendor for proper setup. If supporting your own RTOS and kernel, use the method that best matches your DLL.

The following settings are available:

Vendor This three letter value identifies the RTOS plugin, that is the *.dll file supplied by your vendor.

Load_when Defines when RealView Debugger loads the RTOS plugin:

- load the plugin on connection, with Load_when set to connect
- wait until an RTOS image is loaded, with Load_when set to image_load.

The RTOS features of the debugger are not enabled until the plugin is loaded and has found the RTOS on your target.

Base_address

Defines a base address, overriding the default address used to locate the RTOS data structures. See your RTOS documentation for details.

Exit_Options

Defines how RTOS awareness is disabled. Use the context menu to specify the action to take when an image is unloaded or when you disconnect. You can also specify a prompt.

RSD

Controls whether RealView Debugger enables or disables RSD. This setting is only relevant if your debug target can support RSD.

System_Stop

Use this setting to specify how RealView Debugger responds to a processor stop request when running in RSD mode.

In some cases, it is important that the processor does not stop. This setting enables you to specify this behavior, use:

- Never to disable all actions that might stop the processor.
- Prompt to request confirmation before stopping the processor.
- Don't_prompt to stop the processor. This is the default.

For full details on RTOS support with RealView Debugger, see Chapter 15 *RTOS Support*.

Pre_connect

Forces an order for device connection. When you connect to a device within the .jtg file, this ensures that one or more specific devices are connected first no matter which you selected for connection. This enables pre-setup of the specific devices to guarantee correct operation, such as initializations. You can specify the device(s) to connect to first by name, by processor name, or by processor type, as used in .cnf files that contain target configuration settings.

Commands**Note**

This feature is not supported in this release.

This enables you to specify RealView Debugger commands to run after a connection is established. The most common example is INCLUDE to include commands from a file. The commands are run immediately after the connection is completed. If Pre_connect is set, and the pre-connected device is running this command, the command executes before the original device is connected.

Connect_mode

When you connect to a target, RealView Debugger attempts to establish the connection using the default connect mode, that is No Reset and Stop (see *Setting connect mode* on page 2-11 for details).

Before connecting, RealView Debugger checks to see if a user-defined connect mode has been specified by the Connect_mode setting in your board file, or in any .bcd file linked to the connection. If such a setting is found, it becomes the default connect mode for this connection.

Note

A blank entry in the top-level Connect_mode setting in the board file ensures that any setting in a linked Board/Chip definition file is used instead. This might be important if you are using .bcd files to enable RTOS awareness in RealView Debugger.

Use this setting to specify a connection mode. The options are:

`no_reset_and_stop` Do not submit a reset and halt any process currently running.

`no_reset_and_no_stop`
Do not submit a reset or halt any process currently running.

`reset_and_stop` Do a processor reset and halt any process currently running.

`reset_and_no_stop` Do a processor reset but do not halt any process currently running.

`prompt` Display a prompt for the connection mode to use.

The options available for the Connect_mode setting are generic to all vehicles and supported processors and so might include options that are not supported by your target vehicle.

If you set connect mode from the Connection Control window, using **Connect (Defining Mode)...**, this temporarily overrides any user-defined setting(s) in your target configuration file.

Note

If a prompt is specified in your board file, or in any .bcd file linked to the connection, it takes priority over any other user-defined connect mode setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.

Disconnect_mode

When you disconnect from a target, RealView Debugger attempts to disconnect using the default disconnect mode, that is As-is with Debug (see *Setting disconnect mode* on page 2-20 for details).

Before disconnecting, RealView Debugger checks to see if a user-defined disconnect mode has been specified by the `Disconnect_mode` setting in your board file, or in any `.bcd` file linked to the connection. If such a setting is found, it becomes the default disconnect mode for this connection.

Note

A blank entry in the top-level `Disconnect_mode` setting in the board file ensures that any setting in a linked Board/Chip definition file is used instead. This might be important if you are using `.bcd` files to enable RTOS awareness in RealView Debugger.

Use this setting to specify a disconnection mode. The options are:

- `as_is_with_debug` Leave the target in its current state, whether stopped or running, and maintain any debugging controls.
- `as_is_without_debug` Leave the target in its current state, whether stopped or running, and remove any debugging controls.
- `prompt` Display a prompt for the disconnection mode to use.

The options available for the `Disconnect_mode` setting are generic to all vehicles and supported processors and so might include options that are not supported by your target vehicle.

If you set disconnect mode from the Connection Control window, using **Disconnect (Defining Mode)...**, this temporarily overrides any user-defined setting(s) in your target configuration file.

Note

If a prompt is specified in your board file, or in any `.bcd` file linked to the connection, it takes priority over any other user-defined disconnect mode setting. This prompt-first rule holds true regardless of where the setting is in the configuration hierarchy.

Id_chip

By default, the chip-id, or silicon-id, is loaded from the processor. When accessing special custom chips, it might be necessary to force the ID explicitly. The ID can be expressed as a 16-bit number or in *num.num.num* format. It can also be expressed as a name of the family member if known.

Id_match

This contains the expected silicon ID from the processor. If it does not match this value, you are prompted to choose whether or not to continue the connect operation. The ID can be expressed as a 16-bit number or in *num.num.num* format.

Chip_name

This defines the manufacturer name of the actual device, such as family name or core name. The Chip_name field enables you to specify a name to use in messages and lists displayed by RealView Debugger. It does not enforce the chip family selection. For that, you must use the Id_chip field.

Endianness

This field applies to ARM cores. Use it to set the byte order of the simulated processor.

B.4.2 BOARD, CHIP, and COMPONENT groups

Use these groups to hold connection settings when a standard board or chip, exists. Use a BOARD group to define a target board from a commercial hardware vendor, for example the ARM Evaluator-7T, or a custom design. Similarly, use a CHIP group to define significant devices on a target board or where the device itself is complex. You can use a COMPONENT group to define a core plus ASICs, either commercial or custom.

A CONNECTION group can refer to one or more of these groups by name or ID. A reference to one of these groups enables automatic use of the .jtg file, settings, and ETV information (ASIC, peripherals, and memory).

This type of entry can specify the default connection information which can then be overridden by settings in the CONNECTION group (see *CONNECTION groups* on page B-26 for details).

When you create a new group, the following entries are available, as described in *Generic groups and settings* on page B-5:

- Connect_with
- Advanced_Information

- Configuration
- Description
- Project
- Family_select
- BoardChip_name.

———— **Note** —————

If you create a new BOARD, CHIP, or COMPONENT entry, the Connect_with group also contains an Ethernet group of settings. These are not supported in this release.

B.5 Custom connection reference

To configure custom connections, you must set up one or more of the following groups in the board file:

- *CONNECTION groups*
- *DEVICE groups*

B.5.1 CONNECTION groups

CONNECTION entries are used to get a list of one or more target devices. This setting specifies:

- the type of the device
- the position of the device in the scan chain
- a name used to specify what to connect to.

In some JTAG file forms, additional information such as speed adjust can also be specified. Using a CONNECTION group automatically pulls the list of devices from the named file and provides an easy way to keep the two locked together.

The following entries are available, as described in *Generic groups and settings* on page B-5:

- Connect_with
- Remote
- Advanced_Information
- Configuration
- Auto_connect
- Pre_connect
- Description
- Project
- Disabled
- Shared
- BoardChip_name
- Family_select.

B.5.2 DEVICE groups

Use a DEVICE entry when only one device exists on the scan chain or when you have to specify a lot of information for a specific device. The name of this group must be a name within the .jtg file.

The following entries are available, as described in *Generic groups and settings* on page B-5:

- Connect_with
- Remote
- Advanced_Information
- Description
- Project
- Configuration
- Disabled
- Shared
- Family_select
- BoardChip_name.

You can use a CONNECTION entry instead of a DEVICE entry.

Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

Access-provider connection

A debug target connection item that can connect to one or more target processors. The term is normally used when describing the RealView Debugger Connection Control window.

Address breakpoint A type of breakpoint.

See also Breakpoint.

ADS *See* ARM Developer Suite.

ARM Developer Suite (ADS)

A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors. ADS is superseded by RealView Developer Suite (RVDS).

See also RealView Developer Suite.

ARM instruction A word that encodes an operation for an ARM processor operating in ARM state. ARM instructions must be word-aligned.

ARM state	<p>A processor that is executing ARM instructions is operating in ARM state. The processor switches to Thumb state (and to recognizing Thumb instructions) when directed to do so by a state-changing instruction such as BX, BLX.</p> <p><i>See also</i> Thumb state.</p>
ARM Toolkit Proprietary ELF (ATPE)	<p>The binary file format used by RealView Developer Kit. ATPE object format is produced by the compiler and assembler tools. The ARM linker accepts ATPE object files and can output an ATPE executable file. RealView Debugger can load only ATPE format images, or binary ROM images produced by the fromELF utility.</p> <p><i>See also</i> RealView Developer Suite.</p>
Asynchronous execution	<p><i>Asynchronous execution</i> of a command means that the debugger accepts new commands as soon as this command has been started, enabling you to continue do other work with the debugger.</p>
ATPE	<p><i>See</i> ARM Toolkit Proprietary ELF.</p>
Backtracing	<p><i>See</i> Call Stack.</p>
Big-endian	<p>Memory organization where the least significant byte of a word is at the highest address and the most significant byte is at the lowest address in the word.</p> <p><i>See also</i> Little-endian.</p>
Board	<p>RealView Debugger uses the term <i>board</i> to refer to a target processor, memory, peripherals, and debugger connection method.</p>
Board file	<p>The <i>board file</i> is the top-level configuration file, normally called <code>rvdebug.brd</code>, that references one or more other files.</p>
Breakpoint	<p>A user defined point at which execution stops in order that a debugger can examine the state of memory and registers.</p> <p><i>See also</i> Conditional breakpoint, Default breakpoint, Hardware breakpoint, Software breakpoint, and Unconditional breakpoint.</p>
Call Stack	<p>This is a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.</p>

Conditional breakpoint

A *breakpoint* that has a condition qualifier assigned, and so halts execution when that condition becomes True. The condition normally references the values of program variables that are in scope at the breakpoint location.

See also Breakpoint, Default breakpoint, Hardware breakpoint, Software breakpoint, and Unconditional breakpoint.

Context menu

See Pop-up menu.

Core module

In the context of Integrator, an add-on development board that contains an ARM processor and local memory. Core modules can run stand-alone, or can be stacked onto Integrator motherboards.

See also Integrator.

Current Program Status Register (CPSR)

See Program Status Register.

DCC

See Debug Communications Channel.

Debug Agent (DA)

The Debug Agent resides on the target to provide target-side support for *Running System Debug* (RSD). The Debug Agent can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the *debug communications channel* (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state.

See also Running System Debug.

Debug Communications Channel (DCC)

A debug communications channel enables data to be passed between RealView Debugger and the EmbeddedICE logic on the target using the JTAG interface, without stopping the program flow or entering debug state.

Debug With Arbitrary Record Format (DWARF)

ARM code generation tools generate debug information in DWARF2 format by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).

Default breakpoint

A *breakpoint* that does not have a condition qualifier or an action assigned. A default breakpoint is always an *unconditional breakpoint*.

See also Breakpoint, Conditional breakpoint, Hardware breakpoint, Software breakpoint, and Unconditional breakpoint.

Deprecated

A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features are to be removed in future versions of the product.

Doubleword A 64-bit unit of information.

DWARF *See* Debug With Arbitrary Record Format.

ELF *See* Executable and Linking Format.

Embedded Trace Macrocell (ETM)

A block of logic, embedded in the hardware, that is connected to the address, data, and status signals of the processor. It broadcasts branch addresses, and data and status information in a compressed protocol through the trace port. It contains the resources used to trigger and filter the trace output.

EmbeddedICE logic

The EmbeddedICE logic is an on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

See also IEEE1149.1.

Emulator In the context of target connection hardware, an emulator provides an interface to the pins of a real core (emulating the pins to the external world) and enables you to control or manipulate signals on those pins.

Endpoint connection

A debug target processor, normally accessed through an *access-provider connection*.

ETM *See* Embedded Trace Macrocell.

ETV *See* Extended Target Visibility.

Executable and Linking Format (ELF)

The industry standard binary file format used by RealView Developer Kit. ELF object format is produced by the ARM object producing tools such as armcc and armasm. The ARM linker accepts ELF object files and can output either an ELF executable file, or a partially linked ELF object.

Extended Target Visibility (ETV)

Extended Target Visibility enables RealView Debugger to access features of the underlying target, such as chip-level details provided by the hardware manufacturer or SoC designer.

Floating Point Emulator (FPE)

Software that emulates the action of a hardware unit dedicated to performing arithmetic operations on floating-point values.

FPE *See* Floating Point Emulator.

Halfword A 16-bit unit of information.

Halted System Debug (HSD)

Usually used for RTOS aware debugging, *Halted System Debug* (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. With the target stopped, the debugger presents RTOS information to you by reading and interpreting target memory.

See also Running System Debug.

Hardware breakpoint

A breakpoint that is implemented using non-intrusive additional hardware. Hardware breakpoints are the only method of halting execution when the location is in *Read Only Memory* (ROM). Using a hardware breakpoint often results in the processor halting completely. This is usually undesirable for a real-time system.

See also Breakpoint, Conditional breakpoint, Default breakpoint, Software breakpoint, and Unconditional breakpoint.

HSD

See Halted System Debug.

IEEE 1149.1

The IEEE Standard that defines TAP. Commonly (but incorrectly) referred to as JTAG.

See also Test Access Port

Integrator

A range of ARM hardware development platforms. *Core modules* are available that contain the processor and local memory.

Joint Test Action Group (JTAG)

An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a *Joint Test Action Group* (JTAG) interface port to communicate with processors. For more information see IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

JTAG

See Joint Test Action Group.

JTAG interface unit

A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the processor, for example to the EmbeddedICE logic and the ETM.

Little-endian

Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word.

See also Big-endian.

Pop-up menu

Also known as *Context menu*. A menu that is displayed temporarily, offering items relevant to your current situation. Obtainable in most RealView Debugger windows or panes by right-clicking with the mouse pointer inside the window. In some windows the pop-up menu can vary according to the line the mouse pointer is on and the tabbed page that is currently selected.

Processor core	The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.
Program Status Register (PSR)	Contains information about the current execution context. It is also referred to as the <i>Current PSR</i> (CPSR), to emphasize the distinction between it and the <i>Saved PSR</i> (SPSR), which records information about an alternate processor mode.
PSR	<i>See</i> Program Status Register.
RealView Compilation Tools (RVCT)	RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of <i>RISC</i> processors.
RealView Connection Broker	RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system.
RealView Debugger Trace	Part of the RealView Debugger product that extends the debugging capability with the addition of real-time program and data tracing. It is available from the Code window.
RealView Developer Kit (RVDK)	RealView Developer Kit is a suite of software development applications, together with supporting documentation and examples, that enables you to write and debug applications for the ARM family of <i>RISC</i> processors.
RealView Developer Suite (RVDS)	The latest suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of <i>RISC</i> processors.
RealView ICE (RVI)	A JTAG-based debug solution to debug software running on ARM processors.
RealView ICE Micro Edition (RVI-ME)	A JTAG-based debug tool for embedded systems.
RSD	<i>See</i> Running System Debug.
RTOS	Real Time Operating System.

Running System Debug (RSD)

Used for RTOS aware debugging, *Running System Debug* (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a *Debug Agent* (DA) that resides on the target. The Debug Agent is scheduled along with other tasks in the system.

See also Debug Agent and Halted System Debug.

RVCT

See RealView Compilation Tools.

RVDK

See RealView Developer Kit.

RVDS

See RealView Developer Suite.

RVI

See RealView ICE.

RVI-ME

See RealView ICE Micro Edition.

Scan chain

A scan chain is made up of serially-connected devices that implement boundary-scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain. Processors might contain several shift registers to enable you to access selected parts of the device.

Scope

The range within which it is valid to access such items as a variable or a function.

Script

A file specifying a sequence of debugger commands that you can submit to the command-line interface using the `include` command.

Semihosting

A mechanism whereby I/O requests made in the application code are communicated to the host system, rather than being executed on the target.

Simulator

A simulator executes non-native instructions in software (simulating a core).

Software breakpoint

A *breakpoint* that is implemented by replacing an instruction in memory with one that causes the processor to take exceptional action. Because instruction memory must be altered software breakpoints cannot be used where instructions are stored in read-only memory. Using software breakpoints can enable interrupt processing to continue during the breakpoint, making them more suitable for use in real-time systems.

See also Breakpoint, Conditional breakpoint, Default breakpoint, Hardware breakpoint, and Unconditional breakpoint.

Software Interrupt (SWI)

An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.

SPSR	Saved Program Status Register. <i>See also</i> Program Status Register.
SWI	<i>See</i> Software Interrupt.
Synchronous execution	<i>Synchronous execution</i> of a command means that the debugger stops accepting new commands until this command is complete.
Synchronous starting	Setting several processors to a particular program location and state, and starting them together.
Synchronous stopping	Stopping several processors in such a way that they stop executing at the same instant.
TAP	<i>See</i> Test Access Port.
TAP Controller	Logic on a device which enables access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1. <i>See also</i> Test Access Port and IEEE1149.1.
Target	The target hardware, including processor, memory, and peripherals, real or simulated, on which the target application is running.
Target vehicle	Target vehicles provide RealView Debugger with a standard interface to disparate targets so that the debugger can connect easily to new target types without having to make changes to the debugger core software.
Target Vehicle Server (TVS)	Essentially the debugger itself, this contains the basic debugging functionality. TVS contains the run control, base multitasking support, much of the command handling, and target knowledge, such as memory mapping, lists, rule processing, board file and .bcd files, and data structures to track the target environment.
Test Access Port (TAP)	The port used to access the TAP Controller for a given device. Comprises TCK , TMS , TDI , TDO , and nTRST (optional).
Thumb instruction	One halfword or two halfwords that encode an operation for an ARM processor operating in Thumb state. Thumb instructions must be halfword-aligned.
Thumb state	A processor that is executing Thumb instructions is operating in Thumb state. The processor switches to ARM state (and to recognizing ARM instructions) when directed to do so by a state-changing instruction such as BX, BLX. <i>See also</i> ARM state.

Tracepoint	A tracepoint can be a line of source code, a line of assembly code, or a memory address.
Tracing	The real-time recording of processor activity (including instructions and data accesses) that occurs during program execution.
Trigger	<p>In the context of breakpoints, a trigger is the action of noticing that the breakpoint has been reached by the target and that any associated conditions are met.</p> <p>In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.</p>
TVS	<i>See</i> Target Vehicle Server.
Unconditional breakpoint	<p>A <i>breakpoint</i> that does not have a conditional qualifier assigned, and so always halts execution.</p> <p><i>See also</i> Breakpoint, Conditional breakpoint, Default breakpoint, Hardware breakpoint, and Software breakpoint.</p>
Vector Floating Point (VFP)	A standard for floating-point coprocessors where several data values can be processed by a single instruction.
VFP	<i>See</i> Vector Floating Point.
Watch	A watch is a variable or expression that you require the debugger to display at every step or breakpoint so that you can see how its value changes. The Watch pane is part of the RealView Debugger Code window that displays the watchpoints you have defined.
Watchpoint	In RealView Debugger, this is a hardware breakpoint.
Word	A 32-bit unit of information.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

- AAPCS 15-56
- About this book viii
- Access configuration settings B-12
- Access-provider connections
 - menu 2-7, 2-8
- Access-provider disconnecting
 - menu 2-9
- Access-rule configuration settings
 - B-10
- Access_Method configuration settings
 - B-16
- Access_size configuration settings
 - B-10
- Action** context menu
 - in Resource Viewer window 15-45, 15-46
- Actions
 - specifying for breakpoints 5-49
 - using with condition qualifiers 5-55
- Address configuration settings B-18
- Address ranges
 - specifying 5-7
- Add/Copy/Edit Memory Map
 - dialog 6-5
- Add/Edit Macros
 - dialog 10-8
- Advanced_Information block 12-3, 12-10, B-8
 - Connect mode 2-12
 - Disconnect mode 2-21
- Advanced_Information configuration settings B-6
- Application_Load configuration settings
 - B-9
- ARM Toolkit Proprietary ELF 3-2
- Armulator
 - configuration settings B-19
- ARM_config configuration settings B-18
- Arm_swi_num configuration settings
 - B-19
- ATPE 3-2
- Attachment
 - of threads 15-32
 - status 15-27
- Attributes configuration settings B-10
- Audience, intended viii
- Auto-disconnect
 - in single processor mode 2-19
- Auto-projects
 - changing settings 3-14
 - closing 3-16
 - creating for images 3-13
 - displaying settings 3-13
 - in the Process Control pane 3-12, 3-13
 - see also* Projects
 - properties 3-12, 3-13
 - saving settings 3-15
- Autoscope 4-3
 - in assembly language 4-3
 - red box 4-3
- Auto_connect configuration settings
 - B-6
- Aux_Configuration setting B-6

B

- Backup files
 - automatic 12-15
 - manual 12-15
 - restoring 12-15
- Backups
 - configuration files 13-15
 - files 12-15
 - restoring 12-15
- Banks
 - of registers 7-7, 7-8
- Base configuration settings B-15, B-17
- Base_address configuration settings
 - B-20
- Base_mask configuration settings B-11
- Base_scale configuration settings B-11
- BCD file
 - memory map configuration 6-2
- BCD files 12-3, 12-9
 - assigning to connection 13-6
 - ETV 13-4
 - in search path 12-14
 - linking to connection 13-6
 - location 12-14
- BOARD
 - and CHIP entries 13-13
 - entries in board file 12-9, 13-13
 - linking entries 13-7, 13-9, 13-11
 - settings reference B-24
- Board file 3-2, 12-2
 - about 12-2, 14-2
 - Advanced_Information block
 - configuration settings B-8
 - asterisk 12-7, 12-8, 14-5
 - BOARD groups B-24
 - CHIP groups B-24
 - COMPONENT groups B-24
 - configuration entries B-4, B-8
 - connection entries B-4, B-26
 - disabled entries 14-3
 - disabling entries 14-5
 - duplicate entries 14-4
 - enabled entries 14-3
 - enabling entries 14-4
 - entries 12-6, B-3
 - in the workspace 2-5
 - location 12-2, 13-3
 - merging hardware settings 13-9
 - ordering hardware 13-9, 13-10, 13-20
 - ordering hardware descriptions 13-9, 13-10, 13-20
 - restoring defaults 14-5
 - restoring entries 14-5
 - selecting 2-5
 - settings reference B-3
 - target configuration 13-3
- BoardChip_name configuration settings B-7
- Board/Chip Definitions
 - creating new 13-15
 - .bcd files 13-4
- Book, about this viii
- Break trigger group 15-48
 - changing 15-50, 15-51
 - disappeared 15-48
 - empty 15-48
- Breakpoint units 5-10
 - in Break/Tracepoints pane 5-10
- Breakpoints 5-2
 - Access 5-61
 - actions 5-49, 5-55
 - and memory mapping 5-9, 15-49
 - and stepping 5-3
 - attaching macros 5-52
 - see also* Break trigger group
 - BREAKEXECUTION command 5-61
 - BREAKINSTRUCTION command 5-61
 - breakpoint types 5-2
 - breakpoint units 5-10
 - class 5-41
 - classes 15-52
 - clearing 5-67, 5-68
 - clearing all 5-19, 5-68
 - clearing quickly 5-11
 - colors 15-50, 15-53
 - condition qualifiers 5-48, 5-55
 - conditional 5-4, 5-55
 - conditional in RSD 15-48
 - controlling behavior 5-48, 5-55
 - copying in the Break/Tracepoints pane 5-65
 - creating a favorite 5-71
 - Debug** menu mappings 5-65
 - default 5-4, 5-5
 - deleting 5-68
 - deleting all 5-68
 - disable at cursor 5-19
 - disabling 5-67
 - editing in the Break/Tracepoints pane 5-64
 - enable at cursor 5-19
 - EXEC 5-61
 - function entry point 5-8
 - hardware 5-3, 5-12, 5-16, 5-22
 - hardware in RSD 15-48
 - hardware limitations 5-12
 - HSD 15-47
 - icons 5-10
 - in the Break/Tracepoints pane 5-59, 5-61, 15-53
 - in the Set Address/Data Break/Tracepoint dialog box 15-52, 15-53
 - INSTR 5-61
 - line number references 5-6
 - managing actions 5-49, 5-50
 - managing qualifiers 5-48
 - markers 5-10, 5-61
 - module name references 5-6
 - operations 5-19
 - pass counts 5-34, 5-48
 - Processor exceptions 5-31
 - qualifiers 5-32
 - Read 5-61
 - recording the triggering of 5-6
 - red disc 5-15
 - red icon 5-10
 - removing 5-68
 - removing all 5-68
 - RSD 15-47
 - RTOS support 15-47, 15-49, 15-52, 15-53
 - saving as favorites 5-71, 5-72
 - saving as history 5-70
 - see also* Chaining breakpoints
 - setting 15-49
 - setting hardware tests 5-45
 - setting in disassembly-level view 5-16
 - setting in other ways 5-18, 5-37
 - setting in source-level view 5-15
 - setting in the Memory pane 5-21, 5-37
 - setting in the Stack pane 5-21, 5-37
 - setting on a branch 5-17

- setting on a function 5-16
 - setting on a symbol 5-16
 - setting on an instruction 5-17
 - setting on inline functions 5-10
 - setting on multiple statements 5-10
 - setting on the statement 5-15
 - setting using drag-and-drop 5-18, 5-37
 - software 5-3, 5-12, 5-16
 - standard 5-10, 5-41
 - toggle at cursor 5-19
 - unconditional 5-4
 - using a mask 5-46
 - using a range 5-46
 - using hardware breakpoints 5-43
 - viewing in disassembly-level view 5-10
 - viewing in source-level view 5-10
 - Write 5-61
 - yellow icon 5-10
 - @entry location specifier 5-8
- Browsers**
- accessing 9-2
 - applying a filter 9-7, 9-28
 - browsing C++ classes 9-17
 - browsing functions 9-19
 - browsing variables 9-21
 - Data Navigator pane 9-5
 - filter metacharacters 9-7, 9-28
 - Function List 9-19
 - in RealView debugger 9-2
 - Module/File List 9-23
 - Register List 9-25
 - scope 9-2
 - Symbol Browser pane 9-17
 - using filters 9-27
 - Variable List 9-17, 9-21
- C**
- Call Stack 7-28
 - see also* Stack
 - Cancel button
 - in the Code window 4-2
 - Captive thread 15-32, 15-46
 - Chaining breakpoints 5-29
 - command qualifiers 5-29
 - converting existing breakpoints 5-29
 - Channel viewers 7-10
 - CHIP
 - and BOARD entries 13-13
 - entries in board file 12-9, 13-13
 - settings reference B-24
 - Chip_name configuration settings B-24
 - Clock_speed configuration settings B-19
 - Code patching
 - assembly patching 8-3
 - Code views 3-10, 3-21, 4-3
 - Dsm** tab 3-10, 3-21, 4-3
 - setting disassembly mode 4-5
 - source code tab 3-10, 3-21, 4-3
 - Src** tab 3-10, 3-21, 4-3
 - toggling 4-12
 - Code window
 - attachment 15-27
 - Colors
 - in memory map 6-8
 - in Memory pane 7-16
 - in Register pane 7-4
 - Command line
 - arguments for RealView Debugger 1-2
 - Command line arguments
 - aws 1-2
 - bat 1-3
 - cmd 1-3
 - exec 1-3
 - home 1-3, 12-13
 - inc 1-3
 - init 1-3
 - install 1-4, 12-12, 12-13
 - jou 1-4
 - log 1-4
 - nologo 1-4
 - s 1-4
 - user 1-4, 12-13
 - Commands
 - ADDFILE 15-22
 - in a journal file 4-13
 - logging 4-16
 - logging from your application 4-13
 - pending 4-2
 - purging 4-2
 - queue 4-2
 - queue rules 4-2
 - RELOAD 15-22
 - RTOS support 15-58
 - to the debugger 4-13
 - Commands configuration settings B-21
 - Commands (CLI)
 - CONNECT 2-15
 - Comments from users xiii
 - COMPONENT
 - entries in board file 12-9
 - settings reference B-24
 - Concat_Register configuration settings B-15
 - Concepts
 - RTOS 15-2
 - Conditional breakpoints 5-4
 - Configuration
 - backups 12-14
 - entries 12-9
 - entries in board file B-4, B-8
 - files 12-11
 - Configuration configuration settings B-6
 - Configuration example
 - connect mode 13-30
 - connect mode rules 13-29
 - custom register 13-34
 - disconnect mode 13-30
 - disconnect mode rules 13-29
 - Integrator 13-23
 - memory block 13-39
 - memory map 13-32
 - RVI-ME 14-8
 - RVI-ME with Integrator 13-23
 - setting connection mode 13-30
 - setting disconnection mode 13-30
 - setting memory block 13-39
 - setting memory map 13-32
 - setting registers 13-34
 - stack heap 13-42
 - top_of_memory 13-42
 - Configuration settings
 - Access B-12
 - Access-rule B-10
 - Access_Method B-16
 - Access_size B-10
 - Address B-18
 - Advanced_Information B-6
 - Application_Load B-9

- Armulator B-19
- ARM_config B-18
- Arm_swi_num B-19
- Attributes B-10
- Auto_connect B-6
- Base B-15, B-17
- Base_address B-20
- Base_mask B-11
- Base_scale B-11
- BoardChip_name B-7
- Chip_name B-24
- Clock_speed B-19
- Commands B-21
- Concat_Register B-15
- Configuration B-6
- Config_file B-19
- Connect_mode B-22
- Connect_with B-5
- Cross_trigger B-20
- Description B-6, B-13
- Disabled B-7
- Disconnect_mode B-23
- D_Abort B-18
- Enabled B-19
- Endianess B-24
- Enum B-16
- Enum B-14, B-15, B-16
- Error B-18
- Exit_Options B-21
- Family_select B-7
- FIQ B-18
- Flash_type B-12
- Fpoint_emu B-19
- generic settings B-5
- Gui_name B-14, B-15, B-16
- Heap_base B-18
- Heap_size B-18
- High_mask B-16
- High_name B-16
- High_shift B-16
- Hostname B-6
- Id_chip B-24
- Id_match B-24
- Internal B-10
- IODEVICE B-5
- IRQ B-18
- Length B-12, B-15, B-16, B-17
- Len_mask B-11
- Len_scale B-11
- Len_table B-11
- Load_command B-9
- Load_set_pc B-9
- Load_using B-9
- Load_when B-20
- Low_mask B-16
- Low_name B-16
- Low_shift B-16
- Manufacturer B-5
- Map_rule B-13
- Mask B-13
- Memory_block B-9
- Memory_type B-15
- Method_name B-16
- Names B-14
- On_equal B-13
- ordering 13-20, 15-13, 15-21
- Peripherals B-16
- Port B-6
- Position B-14
- precedence 13-20, 15-13, 15-21
- Pre_connect B-6, B-21
- priority 13-20
- Project B-7
- Properties B-19
- P_Abort B-18
- Read_only B-14, B-15
- RealView ARMulator ISS B-19
- reference B-3
- Register B-13, B-14, B-16
- Register_base B-11
- Register_enum B-14
- Register_length B-11
- Register_Pos_Len B-11
- Register_Window B-17
- Remote B-6
- Reset B-18
- RSD B-21
- RTOS B-20
- Semihosting B-19
- Shared B-7, B-10
- Shared_id B-10
- Signed B-14
- Size B-14
- Speed B-5
- Stack_Heap B-18
- Stack_size B-18
- Start B-12, B-14, B-16
- SWI B-18
- System_Stop B-21
- Thumb_swi_num B-19
- Top_memory B-19
- Trig_in_dis B-20
- Trig_in_ena B-20
- Trig_out_dis B-20
- Trig_out_ena B-20
- Type B-12, B-15, B-16, B-17
- Undefined B-18
- Update_rule B-12, B-13
- Value B-13
- Vector B-19
- Vectors B-18
- Vector_catch B-19
- Vendor B-20
- Volatile B-10, B-13, B-14, B-15
- Wait_states B-12
- Write_only B-15
- Config_file configuration settings B-19
- Connect
 - No Reset and No Stop** 2-13
 - No Reset and Stop** 2-13
 - Reset and No Stop** 2-13
 - Reset and Stop** 2-13
- CONNECT command 2-15
- Connect mode 2-11
 - configuration example 13-30
 - configuration rules 13-29
 - default 2-11
 - default state 2-14
 - defining 2-12
 - defining for a connection 2-12
 - in Advanced_Information block 2-12
 - not honored 2-11, 13-29, 15-20
 - prompting for a connection 2-14
 - prompting for a disconnection 2-24
 - selection box 2-13
 - state options 2-13
 - substitution 2-11, 13-29, 15-20
 - user-defined 2-11, 2-12
- Connecting 2-4, 3-2, 12-2, 12-3, 12-8
 - default files 2-4
 - to a running target, RTOS 15-16, 15-23
 - to targets 2-10
 - unconfigured targets 2-11
 - using names 2-15
 - using numbers 2-15

- CONNECTION
 - entries in board file B-26
- Connection 12-2
 - collapsed in Connection Control window 2-4
 - configuration 12-2
 - creating custom 12-8
 - creating new RTOS-enabled connection 15-8
 - entries in board file B-4, B-26
- Connection Control** button 2-2
- Connection Control window 2-2, 12-3, 12-8
 - collapsing connections 2-4
 - collapsing entries 2-4
 - collapsing vehicles 2-4
 - connect mode 2-11
 - connecting 2-4, 2-10
 - default connections 2-4
 - defining connect mode for a connection 2-12, 2-14
 - defining connect mode for a connection (default) 2-14
 - defining disconnect mode for a connection 2-22, 2-24
 - defining disconnect mode for a connection (default) 2-24
 - Description 2-3
 - disconnect mode 2-20
 - disconnecting 2-19
 - Endpoint connections 2-4
 - entries 2-3
 - expanding entries 2-4
 - expanding vehicles 2-4
 - failing to connect 2-16
 - managing connections 2-4, 2-6
 - Name 2-3
 - tabs 2-2
 - Target vehicles 2-3
 - user-defined connect mode 2-12
 - user-defined disconnect mode 2-21
 - vehicle 12-8
- Connection properties
 - entries 14-2
 - see also* Board file
- Connection Properties window 12-4, 12-6, 15-12
 - Advanced_Information block B-8
 - asterisk 14-3
 - configuration settings reference B-3
 - connection entries B-26
- Connections
 - connect mode using the CLI 2-15
 - connecting using the CLI 2-15
 - failing 2-16
 - in the workspace 2-15
 - named using the CLI 2-15
 - numbered using the CLI 2-15
- Connect_mode configuration settings B-22
- Connect_with configuration settings B-5
- Context
 - controls 7-26
 - see also* Scope
- Cross_trigger configuration settings B-20
- Current thread 15-28, 15-31
 - changing in CLI 15-28
 - CLI commands 15-28, 15-31
 - details in Output pane 15-28
 - in RSD 15-28, 15-31
 - in unattached window 15-31
 - using the **Cycle Threads** button 15-29
- Custom register
 - configuration example 13-34
- Customizing registers 7-10
- Cycle Connections** button
 - Resource Viewer window 15-44
- Cycle Threads** button 15-28
 - unavailable 15-35
- C++
 - browsing classes 9-17
 - classes in the Symbol Browser 9-18
 - finding a class definition 9-18
 - viewing classes 9-18
- D**
- DA
 - Debug Agent 15-4
- Data Navigator pane 9-5
 - filtering 9-6
- Data values
 - saving as favorites 7-39
- DCC 15-5
- Deadlines, scheduling 15-2
- Debug Agent 15-4
 - DCC 15-5
 - disconnect on exit 15-25
 - ICTC 15-5
 - ICTM 15-32, 15-40
 - in thread list 15-32
 - RTOS 15-4, 15-16
 - undefined exceptions catch 15-18
 - viewing status 15-38
- Debug communications channel 15-5
- Debug** menu 4-10
- Debug target
 - bcd configuration files 13-4
 - board file 12-2, 14-2
 - configuration 3-2, 12-2, 13-2, 14-2
 - configuration files 13-5
 - configuring RVI-ME 2-10
 - connect defining mode 2-12, 2-14
 - connect mode 2-11
 - connecting 2-2, 2-4, 2-10, 3-2, 12-2
 - connecting in single processor mode 2-19
 - connection 12-2
 - creating configuration group 13-14
 - creating connection 13-14
 - creating new 13-14
 - creating new group 13-17
 - creating .bcd file 13-14
 - default configuration files 13-2
 - default connections 2-4
 - disconnect defining mode 2-21, 2-22, 2-24
 - disconnect mode 2-20
 - disconnecting 2-18
 - disconnecting and reconnecting 2-25
 - examples configurations 13-22
 - extended visibility 12-3, 12-10
 - failing to connect 2-16
 - hardware 13-2
 - linking target configuration to connection 13-14
 - memory mapping 6-2, 6-7, 6-8
 - resetting processor 3-21, 4-10
 - RV-msg configuration files 13-3
 - troubleshooting configurations 13-46

- troubleshooting connections 2-17, 13-46
- user-defined connect mode 2-11, 2-12
- user-defined disconnect mode 2-21
- Debugger
 - defining in workspace 11-15
 - internal variables 7-9
 - internals 7-9
 - statistics 7-9
- Default breakpoints 5-4
 - overview 5-5
- Default settings directory 12-14
- Description configuration settings B-6, B-13
- DEVICE
 - entries in board file B-26
- Dialog box
 - Add/Copy/Edit Memory Map 6-5
 - Add/Edit Macros 10-8
 - Favorites Chooser/Editor 9-29
 - Fill Memory with Pattern 8-17
 - Find in Files 9-18
 - Flash Memory Control 8-7, 8-8
 - Function List 9-19
 - HW Break if in Range 5-22
 - HW Break if X, then if Y 5-25
 - HW Break on Data Value match 5-27
 - HW While in func/range, Break if X 5-24
 - Interactive Memory Setting 8-11
 - Interactive Register Setting 8-13
 - Load File to Target 3-3
 - Module/File List 9-23, 9-24, 9-26
 - New/Edit Favorite 5-72
 - Register List 9-25
 - RVConfig 2-8
 - Select File to Include Commands
 - from 4-16
 - Select File to Journal to 4-13
 - Select File to log the STDIO to 4-13
 - Select File to Log to 4-13
 - Simple Break if X 5-20
 - Upload/Download file from/to
 - Memory 8-14, 8-19
 - Variable List 9-21
- Directories
 - backups 12-14
 - default settings 12-14
 - examples 1-7
 - Flash examples 1-7
 - home 1-6, 12-13
 - home and the user ID 1-6
 - installation 1-6, 12-12
 - used in RealView Debugger 1-6
- Disabled configuration settings B-7
- Disconnect
 - As-is with Debug** 2-23
 - As-is without Debug** 2-23
- Disconnect mode 2-20
 - configuration example 13-30
 - configuration rules 13-29
 - default 2-20
 - default state 2-24
 - defining 2-22
 - defining for a disconnection 2-22
 - in Advanced_Information block 2-21
 - not honored 2-21, 2-23
 - selection box 2-23
 - state options 2-23
 - substitution 2-21, 2-23
 - user-defined 2-21
- Disconnecting
 - auto-disconnect 2-19
 - by exiting 2-25
 - Code window behavior 2-18
 - confirming 2-25
 - from Connection Control window 2-19
 - project behavior 2-19
 - reconnecting 2-25
 - using the CLI 2-26
 - using the menu 2-19
- Disconnection options, RTOS support 15-24
- Disconnect_mode configuration settings B-23
- Displaying
 - thread information 15-45
- Downloads
 - RTOS Awareness 15-7
- D_Abort configuration settings B-18
- E**
- Enabled configuration settings B-19
- Endianess configuration settings B-24
- Endpoint connections
 - in Connection Control window 2-4
- Enum configuration settings B-16
- Enum configuration settings B-14, B-15, B-16
- Environment variables
 - JOULOG_UNBUF 4-16
 - RVDEBUG_HOME 1-5, 12-13
 - RVDEBUG_INSTALL 12-12, 12-13
 - RVDEBUG_SHARE 1-5, 12-14
- Error configuration settings B-18
- ETV 12-3, 12-10, 13-2
- EXEC
 - hardware breakpoints 5-61
 - in Break/Tracepoints pane 5-61
- Execution controls 4-7
 - submitting commands 4-2
 - using shortcuts 4-9
- Exit_Options configuration settings B-21
- Expression pointer 7-26
- Extended Target Visibility 13-2
 - ETV 13-2
- F**
- Family_select configuration settings B-7
- Favorites
 - breakpoint 5-70, 5-71
 - categories 9-30, 9-32
 - data value 7-39
 - list 9-31
 - managing 9-29
 - watch 7-38
- Favorites Category 9-32
- Favorites Chooser/Editor dialog box
 - categories 9-32
 - components 9-30
 - displaying 9-30
 - overview 9-29
- Favorites List 9-31
- Feedback xiii
- File** menu
 - in Resource Viewer window 15-42
- File search path 12-14
- Files

- board file 3-2, 12-2
 - configuration 12-11
 - including from the **Debug** menu 4-16
 - rvdebug.brd 12-11
 - *.apr 3-12
 - *.aws 11-2, 11-9
 - *.bcd 12-3, 12-9, 12-11
 - *.brd 3-2, 12-11
 - *.ini 11-2, 11-8, 11-12
 - *.prc 12-11
 - *.prj 3-12
 - *.rvc 12-11
 - *.sav 5-70, 5-71, 5-73, 7-37, 7-39, 9-29
 - .bcd for RTOS 15-7, 15-12
 - .brd for RTOS 15-14
 - .dll for RTOS 15-7
 - .rvc 13-3
 - Fill Memory with Pattern dialog 8-17
 - Filtering
 - functions 9-6
 - metacharacters 9-7
 - module naming conventions 5-7
 - modules 9-6
 - syntax rules for 9-6
 - variables 9-6
 - Find in Files
 - C++ Class definition 9-18
 - dialog 9-18
 - FIQ configuration settings B-18
 - Flash
 - checking contents of 16-4
 - directories 1-7
 - examples 1-7
 - see also* FME files
 - setting interactively 8-5
 - Flash Memory Control dialog 8-7, 8-8
 - Flash MEmory files 8-5
 - Flash programming
 - example 13-46
 - troubleshooting 16-5
 - Flash_type configuration settings B-12
 - FLEXlm license 2-16
 - FME files 8-5
 - Force scope 4-4
 - Fpoint_emu configuration settings B-19
 - Frame pointer 7-23, 7-26
 - Function entry point
 - specifying 5-8
 - Function List dialog 9-19
- ## G
- Global options
 - in workspace 11-2, 11-12
 - Glossary Glossary-1
 - Go button
 - see* Run button
 - Go to Cursor button
 - see* Run to Cursor button
 - Go until Return button
 - see* Run until Return button
 - Group Name/Type selector dialog 14-8
 - Group Name/Type selector dialog box 15-9
 - Gui_name configuration settings B-14, B-15, B-16
- ## H
- Halted System Debug
 - see* HSD
 - Hard real-time 15-2
 - Hardware breakpoints
 - see* Breakpoints
 - Heap 6-13
 - see* Memory
 - Heap_base configuration settings B-18
 - Heap_size configuration settings B-18
 - High-level Step Into button
 - in the Code window 4-7
 - High-level Step Over button
 - in the Code window 4-8
 - High_mask configuration settings B-16
 - High_name configuration settings B-16
 - High_shift configuration settings B-16
 - Home
 - see* Directories
 - Hostname configuration settings B-6
 - HSD
 - breakpoints 15-47
 - defined 5-12, 15-3
 - disabling on exit 15-25
 - switching to RSD 15-4, 15-37, 15-43
 - HW Break if in Range dialog 5-22
 - HW Break if X, then if Y dialog 5-25
 - HW Break on Data Value match dialog 5-27
 - HW While in func/range, Break if X dialog 5-24
- ## I
- ICTC 15-5
 - ICTM 15-32, 15-40
 - Id_chip configuration settings B-24
 - Id_match configuration settings B-24
 - Images
 - auto-set PC on load 3-4
 - connecting on loading 3-6
 - debugging multiple images 3-19
 - disconnecting when loaded 3-22
 - load settings in project 3-15
 - loading 3-2, 6-5
 - loading above 0x8000 3-20
 - loading from a user-defined project 3-2
 - loading from the command line 3-6
 - loading from the Load File to Target dialog box 3-3
 - loading from the Process Control pane 3-5
 - loading multiple 3-19
 - loading progress 3-8
 - managing 3-9
 - passing arguments on loading 3-4, 3-7
 - passing arguments on reloading 3-23
 - passing sections on loading 3-4
 - Processor status 3-8
 - quick loading 3-5
 - reloading 3-21, 3-23
 - removing all details 3-22
 - runtime indicator 3-8
 - set PC to entry point on load 3-5
 - setting PC manually 3-20

- setting PC on load 3-4
- symbol data 3-18
- unloading 3-21
- unloading and auto-projects 3-22
- viewing in the Code window 3-9
- viewing in the Process Control pane 3-10
- IMP Comms Target Controller 15-5
- IMP Comms Target Manager 15-32, 15-40
- Include files 4-16
 - from the **Debug** menu 4-16
 - Journal files 4-13
 - Log files 4-13
 - STDIolog files 4-13
- Information about threads 15-45
- INSTR
 - in Break/Tracepoints pane 5-61
 - software breakpoints 5-61
- Integrator
 - configuration example 13-23
- Intended audience viii
- Interactive Memory Setting
 - dialog 8-11
- Interactive Register Setting
 - dialog 8-13
- Internal configuration settings B-10
- Internationalization A-8
 - settings for A-9
- Interrupts
 - when loading to an RTOS 15-25
- IODEVICE configuration settings B-5
- IRQ
 - when loading to an RTOS 15-25
- IRQ configuration settings B-18

J

- JOULOG_UNBUF 4-16
- Journal files
 - at start-up 4-15, 4-16
 - closing 4-14
 - in the Status line 4-14
 - output buffering 4-16
 - viewing 4-15

L

- Length configuration settings B-12, B-15, B-16, B-17
- Len_mask configuration settings B-11
- Len_scale configuration settings B-11
- Len_table configuration settings B-11
- License
 - FLEXlm 2-16
- Line numbers
 - setting breakpoints 5-15
 - turning on 3-10, 4-4, 4-8, 5-14, 7-2, 8-10, 10-8
- Linker command files
 - generating 6-14
- Linking
 - a board to a connection 13-7
 - avoiding conflicts when 13-6
 - boards to a connection 13-9
 - boards to boards 13-11
- Load File to Target
 - dialog 3-3
- Loading
 - appending 3-4
 - images 3-2, 6-5
 - images and loading progress 3-8
 - images and Processor status 3-8
 - images and runtime indicator 3-8
 - multiple images 3-19
 - replace image 3-4
- Load_command configuration settings B-9
- Load_set_pc configuration settings B-9
- Load_using configuration settings B-9
- Load_when configuration settings B-20
- Localization
 - see* Internationalization
- Log files
 - at start-up 4-15, 4-16
 - closing 4-14
 - in the Status line 4-14
 - output buffering 4-16
 - viewing 4-15
- Low-level Step Into button
 - in the Code window 4-8
- Low-level Step Over button
 - in the Code window 4-8
- Low_mask configuration settings B-16
- Low_name configuration settings B-16

Low_shift configuration settings B-16

M

- Macros 10-2
 - attaching 10-7
 - calling 10-5, 10-14
 - calling from the **Debug** menu 10-14
 - copying 10-13
 - creating 10-8
 - declaring variables 10-9
 - defining 10-4
 - deleting 10-14
 - editing 10-11
 - in symbol table 10-2, 10-5, 10-9, 10-14
 - in the symbol table 3-18
 - including a definition file 10-4, 10-14
 - loading from a project 10-15
 - loading on connection 10-15
 - precedence 10-5
 - properties 10-3
 - return values 10-5
 - stopping 10-7
 - using debugger commands 10-3, 10-9
 - using with breakpoints 5-52
 - viewing 10-10
- Manufacturer configuration settings B-5
- Mapping
 - source files 4-20
- Map_rule configuration settings B-13
- Margin
 - breakpoint markers 5-10
 - red icon 5-10
 - setting breakpoints 5-15
 - white icon 5-10
 - yellow icon 5-10
- Mask configuration settings B-13
- Memory
 - and breakpoints 5-9
 - changing contents 7-17, 7-19
 - data formats 7-15
 - data sizes 7-14
 - disabling memory mapping 6-4
 - display colors 7-16

- downloading to a file 8-14, 8-18
- editing map entries 6-5
- enabling memory mapping 6-4
- filling with a pattern 8-16
- formatting options 7-13
- generating linker command files 6-14
- interactive operations 8-2
- interactive operations examples 8-10
- interactive operations from the **Debug** menu 8-3
- interactive operations from the Memory pane 8-4
- interactive operations from the Stack pane 8-4
- map configuration 6-8
- mapping 6-2
- mapping and breakpoints 5-9
- setting access size 6-6, 7-19
- setting configuration example 13-39
- setting interactively 8-10
- setting stack and heap 6-13
- setting top_of_memory 6-13
- specifying range 8-15
- verifying against a file 8-16
- viewing contents 7-11
- viewing the map 6-7
- working with Flash 8-5
- Memory and breakpoints 15-49
- Memory block
 - configuration example 13-39
- Memory map
 - in BCD file 6-2
 - configuration example 13-32
 - display colors 6-8
 - editing 6-5
 - setting up 6-5
 - update based on processor registers 6-12
- Memory_block configuration settings B-9
- Memory_type configuration settings B-15
- Menus
 - Break/Tracepoints** 5-62, 5-63
 - C++ **Class** 9-18
 - C++ **Function** 9-18

- Debug** 4-10
- Disassembly View** 4-5
- Memory/Register Operations** 8-3
- New Actions** 5-50
- New Qualifiers** 5-48
- Pane** 5-65
- Target** 3-3
- Tools** 11-8
- View** 3-5
- Workspace** 11-3
- Menus (Resource Viewer window)
 - Action** context 15-45, 15-46
 - File** 15-42
 - RSD** 15-43
 - View** 15-42
- Metacharacters
 - in browser filters 9-7, 9-28
 - using in browsers 9-7, 9-28
- Method_name configuration settings B-16
- Modifying
 - local variables 15-56
 - shared variables 15-56
- Module naming conventions 5-7
- Module/File List
 - dialog 9-23, 9-24, 9-26
- Multibyte characters
 - displaying in the **Locals** tab 7-31
 - displaying Watch variables 7-35, 7-36
 - see* Internationalization

N

- Names configuration settings B-14
- New/Edit Favorite
 - dialog 5-72
- Non-captive thread 15-31
- Numbers
 - turn on lines 3-10, 4-4, 4-8, 5-14, 7-2, 8-10, 10-8

O

- Objects
 - closing open 11-5
- On_equal configuration settings B-13

- Options window 11-8
- OS marker
 - context menu 15-37
 - error status 15-37
 - HSD in the Process Control pane 15-35
 - in the Process Control pane 15-35, 15-36
 - RSD in the Process Control pane 15-36
 - status and meaning 15-36

P

- Panes
 - Break/Tracepoints 5-11, 5-59, 5-60, 15-53
 - Call Stack 7-28
 - Data Navigator 9-5
 - File Editor 4-3
 - Memory 7-11
 - Output 15-28
 - Process Control 3-10, 15-34, 15-35, 15-36, 15-38
 - Register 7-2
 - Stack 7-23, 7-25
 - Symbol Browser 9-17
 - Watch 7-32
- Pass counts
 - used in hardware breakpoints 5-34
 - used in software breakpoints 5-48
- PC 3-4
 - see* Program Counter
- Pending
 - commands 4-2
- Peripherals configuration settings B-16
- Persistence info
 - breakpoints history 5-70
 - favorites 5-71, 7-37
- Plugins
 - for RTOS 13-4, 15-7
- Port configuration settings B-6
- Position configuration settings B-14
- Pre_connect configuration settings B-6, B-21
- Process
 - and threads 15-3

- in Process Control pane 3-11, 15-34
 - multiple images 3-19
 - Process Control
 - properties 3-12
 - tabs 3-10
 - Process Control pane
 - Process** tab 15-34
 - Thread** tab 15-38
 - type ahead 3-12
 - working with threads 15-34
 - Processor
 - reset 3-21
 - resetting 3-21
 - Processor exceptions
 - see* Breakpoints
 - Processor mode
 - in register view 7-7
 - privileged mode 7-7
 - Program Counter 3-4, 4-3
 - locating 4-4
 - red box 4-3, 4-4
 - reset to entry point 4-5
 - see also* Scope
 - setting on load 3-4
 - Project configuration settings B-7
 - Projects
 - associated images 3-16
 - auto-projects 3-12, 3-13
 - closing 3-16
 - defining in workspace 11-6
 - on disconnecting 2-19
 - Project Properties window 4-19
 - searching source files 4-17
 - source files mapping 4-20
 - Properties configuration settings B-19
 - P_Abort configuration settings B-18
- Q**
- Queue
 - command handling 4-2
 - commands 4-2
- R**
- Read_only configuration settings B-14, B-15
 - RealMonitor
 - plugins 13-4
 - RealView ARMulator ISS
 - configuration settings B-19
 - RealView Debugger
 - arguments 1-2
 - connecting to a target 3-2, 12-2
 - debugger internals 7-9
 - directories 1-6
 - image loading 6-5
 - memory mapping 6-2
 - starting 1-2
 - target configuration 3-2, 12-2
 - using macros 10-2
 - Register configuration settings B-13, B-14, B-16
 - Register List
 - dialog 9-25
 - Registers
 - ARM processors 7-7
 - banked out 7-7, 7-8
 - banks 7-7, 7-8
 - changing contents 7-6
 - defining custom registers 7-10
 - display colors 7-4
 - formatting options 7-4
 - interactive operations 8-2
 - interactive operations examples 8-10
 - interactive operations from the **Debug** menu 8-3
 - processor mode 7-7
 - setting configuration example 13-34
 - setting interactively 8-12
 - updating contents 7-7
 - viewing 7-7
 - viewing contents 7-2
 - viewing debugger internals 7-9
 - viewing for threads 15-56
 - Register_base configuration settings B-11
 - Register_enum configuration settings B-14
 - Register_length configuration settings B-11
 - Register_Pos_Len configuration settings B-11
 - Register_Window configuration settings B-17
 - Reloading
 - images 3-23
 - using arguments 3-23
 - Remote configuration settings B-6
 - Reset configuration settings B-18
 - Resource Viewer window 15-41, 15-42, 15-45
 - Action** context menu 15-45
 - Action** context menu parameters 15-45
 - action parameters 15-45
 - Conn** tab 15-42
 - Details area 15-45
 - File** menu 15-42
 - Resources list 15-42, 15-45
 - Resources toolbar 15-44
 - RSD** menu 15-43
 - RTOS plugin 15-45
 - tabs in 15-42, 15-45
 - View** menu 15-42
 - Resources toolbar
 - in Resource Viewer window 15-44
 - RSD
 - breakpoints 15-18, 15-47
 - conditional breakpoints 15-48
 - defined 15-4
 - disable configuration setting 15-16
 - disabling on exit 15-25
 - enable configuration setting 15-16
 - hardware breakpoints 15-48
 - switching to HSD 15-4, 15-37, 15-43
 - system breakpoints 15-47
 - thread breakpoints 15-47
 - RSD configuration settings B-21
 - RSD** menu (Resource Viewer window) 15-43
 - RTOS x, 15-1, 15-2
 - actions on objects 15-45
 - Advanced_Information block 15-14
 - ARM_config configuration settings 15-17
 - Base_address configuration setting 15-15
 - Board/Chip definition files 13-4, 15-7
 - break trigger group 15-48

- Breakpoint Classes 15-52
 - breakpoints 15-47, 15-49, 15-50, 15-52, 15-53
 - Break/Tracepoints pane 15-53
 - CLI commands 15-58
 - concept 15-2
 - conditional breakpoints 15-48
 - configuration settings 15-14, 15-24
 - configuration settings conflicts 15-16
 - configuring an RTOS-enabled connection 15-12
 - Connect mode configuration rules 15-18, 15-19
 - connecting to a running target 15-23
 - connecting to a target 15-22
 - Connect_mode configuration settings 15-18, 15-19, 15-23
 - Cpu_id configuration setting 15-15
 - creating a new RTOS-enabled connection 15-8
 - current thread 15-28
 - Cycle Threads** button 15-28
 - data structures 15-42
 - DCC 15-5
 - Debug Agent 15-4, 15-16
 - Disconnectmode configuration rules 15-18, 15-19
 - disconnection configuration settings 15-24
 - disconnection options 15-24
 - Disconnect_mode configuration settings 15-18, 15-19, 15-24
 - Exit_Options configuration setting 15-16, 15-24
 - Exit_Options prompt 15-24
 - formatting memory 15-57
 - hardware breakpoints 15-48
 - HSD 15-3
 - HSD breakpoints 15-47
 - ICTC 15-5
 - ICTM 15-32, 15-40
 - interactions with debugger 15-46
 - interrupts when loading 15-25
 - IRQ 15-25
 - jobs 15-2
 - loading images 15-25, 15-26
 - loading the plugin 15-15
 - Load_when configuration setting 15-15, 15-16
 - Map** tab 15-34
 - objects 15-45
 - OS marker 15-35, 15-36
 - OS state 15-25
 - plugins 13-4, 15-7
 - Process** tab 15-34, 15-35
 - Real-time applications 15-2
 - resetting OS state when reloading 15-25
 - Resource Viewer window 15-41, 15-42, 15-44
 - resuming threads 15-46
 - RSD 15-4
 - RSD breakpoints 15-47
 - RSD configuration setting 15-16
 - RTOS Awareness Downloads 15-7
 - RTOS configuration settings 15-14
 - see also* HSD
 - Set Address/Data Break/Tracepoint dialog box 15-52, 15-53
 - special threads in the **Thread** tab 15-40
 - specifying image target name 15-26
 - stepping threads 15-54
 - stopping threads 15-46
 - suspending threads 15-46
 - System_Stop configuration setting 15-16, 15-17
 - Thread list 15-29
 - thread status 15-46
 - Thread** tab 15-34, 15-38
 - threads in the **Thread** tab 15-39
 - Vendor configuration setting 15-15
 - viewing registers 15-56
 - RTOS Awareness 15-7
 - RTOS configuration settings B-20
 - RTOS resources
 - interactions with debugger 15-46
 - Run** button
 - in the Code window 4-7
 - Run to Cursor** button
 - in the Code window 4-9
 - Run until Return** button
 - in the Code window 4-9
 - Run-time stack
 - see* Stack
 - RVConfig
 - Advanced settings 14-11
 - and .rvc files 12-11
 - with RVI-ME 14-10
 - RVConfig dialog 2-8
 - RVDEBUG_HOME 1-5, 12-13
 - RVDEBUG_INSTALL 12-12, 12-13
 - RVDEBUG_SHARE 1-5, 12-14
 - RVI-ME
 - adding targets 14-7, 15-9
 - configuration files 12-16
 - configuring devices 14-10
 - configuring interface unit 14-10
 - configuring new connection 14-8, 15-8
 - configuring the target 2-5, 2-8, 2-17
 - Connect mode 2-11, 13-29, 15-20
 - Disconnect mode 2-11, 13-29, 15-20
 - in the Connection Control window 14-7
 - Integrator configuration example 13-23
 - RVConfig 2-8, 14-10
 - viewing targets 14-7
- RV-msg
 - configuration files 13-3
- ## S
- Scheduling to deadlines 15-2
 - Scope 4-3
 - autoscope 4-3
 - blue pointer 4-4
 - context 4-3
 - controls 7-26, 7-31
 - filled blue arrow 4-4, 4-5
 - forcing 4-4
 - in browsers 9-2
 - PC 4-3
 - red box 4-4, 4-5, 4-7
 - Scoping
 - to a file 9-24, 9-26
 - to a function 9-20
 - to a module 9-24, 9-26
 - Search path, file 12-14
 - Search rules 4-17, 4-18
 - autoconfiguring 4-18
 - configuring 4-19

- Searching
 - source files 4-17, 4-20
 - source search rules 4-17
 - Select File to Include Commands from dialog 4-16
 - Select File to Journal to dialog 4-13
 - Select File to log the STDIO to dialog 4-13
 - Select File to Log to dialog 4-13
 - Semihosting configuration settings B-19
 - Settings
 - in Workspace A-1
 - save on exit 11-4
 - Settings options 11-4
 - Shared configuration settings B-7, B-10
 - Shared_id configuration settings B-10
 - Signed configuration settings B-14
 - Simple Break if X dialog 5-20
 - Size configuration settings B-14
 - Soft real-time 15-2
 - Software breakpoints
 - see* Breakpoints
 - Source coloring
 - settings for A-8, A-9
 - Source search prompt 4-18
 - Speed configuration settings B-5
 - Stack
 - breakpoints 7-26
 - changing contents 7-25
 - context controls 7-26, 7-31
 - data formats 7-25
 - EP 7-26
 - formatting options 7-24
 - FP 7-23, 7-26
 - in the Stack pane 7-22, 7-23
 - see also* Memory
 - pointer 7-22
 - SP 7-23, 7-26
 - Stack down button
 - in the Code window 7-26
 - Stack heap
 - configuration example 13-42
 - Stack pointer 7-23, 7-26
 - default 7-26
 - Stack up button
 - in the Code window 7-26
 - Stack_bottom configuration settings B-18
 - Stack_Heap configuration settings B-18
 - Stack_size configuration settings B-18
 - Standard breakpoint 5-10, 5-41
 - Start configuration settings B-12, B-14, B-16
 - Starting
 - RealView Debugger 1-2
 - STDIolog files
 - at start-up 4-15, 4-16
 - closing 4-14
 - in the Status line 4-14
 - output buffering 4-16
 - viewing 4-15
 - Stop Execution button
 - in the Code window 4-7
 - Structure of this book viii
 - SWI configuration settings B-18
 - Symbol Browser pane 9-17
 - Symbols
 - loading 3-4, 3-18
 - macros in the symbol table 3-18
 - symbol table 3-18
 - Synch** tab
 - Connection Control window 2-2
 - System breakpoints (RSD) 15-47
 - System_Stop configuration settings B-21
- T**
- Target
 - see* Debug target
 - Target vehicle
 - in Connection Control window 2-3
 - Task Control Block 15-56
 - TCB, used by threads 15-56
 - Terminology Glossary-1
 - Thread
 - actions on 15-45
 - and processes 15-3
 - attachment 15-27, 15-32, 15-39
 - captive 15-32, 15-46
 - changing the current thread 15-29
 - changing variables 15-57
 - Color Box 15-29
 - current 15-28, 15-31
 - Cycle Threads** button 15-28
 - details of 15-30
 - displaying the thread list 15-28, 15-30
 - grayed in thread list 15-32
 - HSD 15-3
 - in attached window 15-31
 - in the thread list 15-28, 15-30
 - list 15-28, 15-29, 15-30
 - markers 15-39
 - non-captive 15-31
 - not controlled by RealView Debugger 15-32
 - registers 15-56
 - Resource Viewer window 15-41
 - resuming 15-46
 - RSD 15-4
 - selecting a new thread 15-29
 - selection box 15-33
 - special threads 15-40
 - status 15-46
 - stepping 15-54
 - stepping in attached window 15-55
 - stepping in background 15-55
 - stepping in unattached window 15-55
 - stopping 15-46
 - suspending 15-46
 - unattached windows 15-27
 - updating memory views 15-57
 - updating views 15-57
 - updating watches 15-57
 - viewing in Code window 15-31
 - Thread breakpoints
 - RSD 15-47
 - stepping 15-54
 - Thread list 15-29
 - cycling 15-29
 - Thread** tab 15-38
 - Debug Agent 15-40
 - special threads 15-40
 - threads view 15-39
 - unavailable 15-35
 - Threads
 - in the Break/Tracepoints pane 5-60, 5-64
 - see also* Halted System Debug

Thumb_swi_num configuration settings B-19

Toolbars
 customizing 11-12

Top_memory configuration settings B-19

top_of_memory
 configuration example 13-42

Tracepoints 4-12
 class 5-41
 clearing 5-68
 clearing all 5-19, 5-68
 deleting 5-68
 deleting all 5-68
 disabling 5-67
 in the Break/Tracepoints pane 5-59, 5-60, 5-64
 removing 5-68
 removing all 5-68
 type 5-41

Trig_in_dis configuration settings B-20

Trig_in_ena configuration settings B-20

Trig_out_dis configuration settings B-20

Trig_out_ena configuration settings B-20

Troubleshooting
 configurations 13-46
 connections 2-17
 Flash programming 16-5

Type configuration settings B-12, B-15, B-16, B-17

U

Unconditional breakpoints 5-4

Undefined configuration settings B-18

Unloading
 images 3-21

Update_rule configuration settings B-12, B-13

Upload/Download file from/to Memory dialog 8-14, 8-19

User ID
 creating the home directory 1-6

User÷Ps comments xiii

V

Value configuration settings B-13

Variable List
 dialog 9-21

Variables, changing in threads 15-57

Vector configuration settings B-19

Vectors configuration settings B-18

Vector_catch configuration settings B-19

Vehicle 12-8
 connecting to RVI-ME 12-8
 menu 2-6

Vendor configuration settings B-20

View menu
 in Resource Viewer window 15-42

Volatile configuration settings B-10, B-13, B-14, B-15

W

Wait_states configuration settings B-12

Watches
 creating a favorite 7-38
 editing 7-37
 in the Watch pane 7-32
 managing 7-34
 saving as favorites 7-38
 setting 7-32
 setting from favorites 7-37
 viewing expressions 7-33

Windows
 Build-Tool Properties 15-8
 Connection Control 2-2, 12-3, 12-8
 Connection Properties 12-4, 12-5
 Options 11-8
 Project Properties 4-19
 Resource Viewer 15-41
 Workspace Options 11-8

Windows login ID
 creating the home directory 1-7

Workspace
 ALL group 11-13, A-8
 see also Appendix A
 Asm_type setting A-7
 Backup group A-12
 Backup settings A-12

Board_file setting A-5

Button group A-6

Button settings A-6

changing settings 11-14

closing 11-5

closing objects 11-5

Cmds group A-13

CODE group 11-12, A-6

Command group A-2

Command settings A-2

connection details 11-13

copying entries 11-16

creating an empty file 11-7

cutting entries 11-18

DEBUGGER group 11-12, A-2

debugger settings 11-15

defining Code windows 11-15

deleting settings 11-18

Disassembler group A-3

Disassembler settings A-3

Edit group A-12, A-14

Edit settings A-14

Font_information group A-9

Font_information settings A-9

global options 11-2, 11-12

in **Tools** menu 11-8

in Workspace Options window 11-8

initializing 11-2

Internationalization group A-9

Internationalization settings A-9

List of Entries pane 11-10

not using 11-2

open projects 11-6

opening 11-3, 11-5

Pane_font settings A-9

pasting entries 11-16

Pos_size group A-6

Pos_size settings A-6

project load list 11-6

PROJECTS group 11-13

resetting entries 11-18

restoring settings 11-14

same on start-up 11-4

saving 11-3

Search group A-11

Search settings A-11

settings 11-2, 11-12

settings conflict 11-5, 11-12

settings file 11-12

- settings saved 11-2
- Settings Values pane 11-10
- Source_coloring group A-9
- Source_coloring settings A-10
- Src_ctrl group A-13
- Src_ctrl settings A-13
- start-up options 11-3
- Tab_conv group A-13
- Tab_conv settings A-13
- text color settings A-10
- Text group A-8
- text Height setting A-8
- text Internationalization setting A-8
- text Src_color_dis setting A-8
- text Width setting A-8
- undoing changes 11-18
- using 11-2
- viewing settings 11-8
- window internals 11-13
- Workspace** menu
 - in the Code window 11-3
- Workspace Options window 11-9
- Write_only configuration settings B-15

Symbols

- @entry
 - breakpoint location specifier 5-8