

RealView[®] Developer Kit

Version 2.2

Extensions User Guide



RealView Developer Kit

Extensions User Guide

Copyright © 2006 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
January 2006	A	Release for RVDK v2.2 with Trace

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 Extensions User Guide

Preface

About this book	vi
Feedback	ix

Chapter 1

Introduction to RealView Debugger Extensions

1.1	About RealView Debugger extensions	1-2
1.2	Licensing	1-5
1.3	Supported platforms	1-6
1.4	Supported hardware	1-7

Chapter 2

Tracing with RealView Debugger

2.1	About tracing with RealView Debugger	2-2
2.2	Getting started	2-8
2.3	Configuring the ETM	2-16
2.4	Tracepoints in RealView Debugger	2-29
2.5	Configuring trace capture	2-35
2.6	Setting unconditional tracepoints	2-39
2.7	Setting conditional tracepoints	2-48
2.8	Managing tracepoints	2-68
2.9	Capturing trace output	2-72
2.10	Using the Analysis window	2-73
2.11	Mapping Analysis window options to CLI commands and qualifiers	2-130

2.12	Examples of using trace in RealView Debugger	2-134
------	--	-------

Chapter 3

RTOS Support

3.1	About Real Time Operating Systems	3-2
3.2	Using RealView Debugger RTOS extensions	3-7
3.3	Connecting to the target and loading an image	3-22
3.4	Associating threads with views	3-27
3.5	Working with OS-aware images in the Process Control pane	3-34
3.6	Using the Resource Viewer window	3-41
3.7	Debugging your RTOS application	3-47
3.8	Using CLI commands	3-58

Appendix A

Setting up the Trace Hardware

A.1	ARM RealView Trace and RealView ICE	A-2
-----	---	-----

Appendix B

Setting up the Trace Software

B.1	ARM RealView Trace and RealView ICE	B-2
-----	---	-----

Glossary

Preface

This preface introduces the *RealView® Developer Kit v2.2 Extensions User Guide*. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page ix.

About this book

This book describes how to use the following RealView Debugger extensions:

- the RealView Debugger tracing extension
- the RealView Debugger *Real Time Operating System* (RTOS) extension, which requires additional software support from the RTOS vendor.

This book only describes the debugger extensions. See the other books in the RealView Developer Kit documentation suite for more information about the debugger.

Intended audience

This book has been written for users of the RealView Debugger extensions. It is assumed that users are experienced programmers.

Although prior experience of using RealView Debugger is not assumed, it is recommended that users first familiarize themselves with performing common debugging operations before using the extensions. The technical level of the audience is assumed to be relatively high. Depending on the RealView Debugger extension being used, the following additional experience is recommended:

Tracing and profiling

You must understand how real-time tracing is beneficial in helping to debug programs that are running at full clock speed.

RTOS support

You must have some experience with debugging an RTOS application.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction to RealView Debugger Extensions*

Read this chapter for a general overview of the RealView Debugger extensions, for system requirements that are applicable to each extension, and for details on the structure of this book.

Chapter 2 *Tracing with RealView Debugger*

Read this chapter for a description of the support RealView Debugger provides for tracing, including how to generate trace data using RealView Debugger, and how to analyze the trace output using the Analysis window.

Chapter 3 RTOS Support

Read this chapter for a description of the support RealView Debugger provides for debugging an RTOS application.

Appendixes and Glossary

Appendix A *Setting up the Trace Hardware*

Read this appendix for details on how to set up the hardware for the trace configurations supported by RealView Debugger.

Appendix B *Setting up the Trace Software*

Read this appendix for details of how to set up the software for the trace configurations supported by RealView Debugger.

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 on developing code for the ARM® family of processors.

ARM Limited periodically provides updates and corrections to its documentation. See the Documentation area of <http://www.arm.com> for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

ARM publications

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

- *RealView Developer Kit v2.2 Debugger User Guide* (ARM DUI 0281)
- *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 Guide* (ARM DUI 0284).
- *RealView Developer Kit v2.2 Linker and Utilities Guide* (ARM DUI 0285)
- *RealView Developer Kit v2.2 Project Management User Guide* (ARM DUI 0291).

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

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

See the *RealView ICE User Guide* (ARM DUI 0155) for information relating to the ARM debug interfaces suitable for use with RealView Debugger.

Other publications

For a comprehensive introduction to ARM architecture see:

Steve Furber, *ARM System-on-Chip Architecture, Second 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, second 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 (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

Introduction to RealView Debugger Extensions

This chapter introduces the RealView® Debugger extensions that are available, and shows how you can find more information on these extensions throughout this book. It contains the following sections:

- *About RealView Debugger extensions* on page 1-2
- *Licensing* on page 1-5
- *Supported platforms* on page 1-6
- *Supported hardware* on page 1-7.

1.1 About RealView Debugger extensions

In addition to the main RealView Debugger functionality, there are several extensions available to users. Some extensions are only available if you have the appropriate license.

This section introduces the chapters in this book that fully describe the RealView Debugger extensions:

- *Chapter 2 Tracing with RealView Debugger*
- *Chapter 3 RTOS Support* on page 1-3.

Note

This book describes only the RealView Debugger extensions. It does not provide details on using RealView Debugger for common debugging tasks. For complete details on using RealView Debugger, see the *RealView Developer Kit v2.2 Debugger User Guide*.

1.1.1 Chapter 2 Tracing with RealView Debugger

This chapter describes how to use RealView Debugger to generate trace information, and how to analyze trace results using the Analysis window. You perform tracing using trace hardware. Using trace hardware with a processor that contains an *Embedded Trace Macrocell™* (ETM) enables you to use the widest range of tracing and analysis features.

You can set trace capture details by setting either:

- simple trace control points, ranges, and triggers
- conditional tracepoints that can include conditions similar to the types of conditions you can set for breakpoints.

After you have generated trace information, you can then use the Analysis window to:

- configure tracing options to apply to all trace captures you perform
- view trace and profiling information, using different views
- filter the results of a captured trace
- search for a specific item of trace information
- manipulate the display of trace information, and includes:
 - the display of inferred register values, interleaved with the trace output
 - the summing of profiling data over multiple runs of your application.

Note

It is recommended that you read *Examples of using trace in RealView Debugger* on page 2-134 before attempting to perform tracing on your own program. This section provides examples of using trace to solve typical development problems. It assumes you have prior experience with using RealView Debugger.

1.1.2 Chapter 3 RTOS Support

This chapter describes the RTOS support available for developers using RealView Debugger on Windows, and the benefits and limitations of that support.

RTOS-specific files are not installed with RealView Debugger. RTOS awareness is enabled by using plugins supplied by your RTOS vendor. This means that you must download the files you require after you have installed RealView Debugger. For more information, select the following option from the Code window menu:

Help → ARM on the Web → Goto RTOS Awareness Downloads

RTOS debugging modes

RealView Debugger supports different debugging modes:

Halted System Debug

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

HSD is available through part of the RTOS plugin.

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 the whole of your debug target before carrying out any analysis of your system. However, you might have to stop a thread or group of threads if you want to:

- change variables for a specific thread
- view the static state of a thread, because the Resource Viewer window shows only a snapshot of transient data.

RSD is only available where supported by your debug target. It relies on having an RTOS plugin installed that includes support for RSD and the target Debug Agent.

Where RSD is supported, RealView Debugger enables you to switch seamlessly between RSD and HSD mode using Code window controls or CLI commands (as described fully in this chapter).

RTOS debugging facilities available in RealView Debugger

Chapter 3 shows how to use the Code window, and the additional tabs available in the Resource Viewer window, to debug an RTOS application. Using these facilities, you can:

- attach and detach threads to Code windows, enabling you to monitor one or more threads in the system
- select individual threads to display the registers, variables, and code related to that thread
- change the register and variable values for individual threads
- display additional information for kernel resources.

———— Note ————

It is recommended that you read this chapter before attempting to debug an RTOS application using RealView Debugger. This chapter includes debugging examples and assumes you have experience with using RealView Debugger only for single-threaded programs.

1.2 Licensing

You do not require a separate license to use Trace.

The RTOS extension is not licensed by ARM Limited, but you must obtain enabling software for your chosen RTOS (see Chapter 3 *RTOS Support* for more information).

1.3 Supported platforms

The RealView Debugger licensed extensions are supported on the same platforms as the RealView Debugger itself is supported. See your installation notes for a list of supported platforms.

Note

Be aware of the following:

- With the exception of the RTOS extension, there are no additional software requirements to use the RealView Debugger extensions.
 - RTOS support is currently only available for developers working on Windows workstations.
-

1.4 Supported hardware

The type of hardware target that is supported by the RealView Debugger extensions is dependent on the extension you are using:

- *Hardware for tracing*
- *Hardware for RTOS support.*

1.4.1 Hardware for tracing

The Trace extension to RealView Debugger requires you to use an ETM-enabled ARM® processor, or any other supported processor with an on-chip trace buffer capability so that you can view the contents of a trace buffer. For details on the processors supported by the tracing extension, see *Requirements for tracing* on page 2-2.

In addition, the tracing extension supports the use of several trace hardware configurations. See Appendix A *Setting up the Trace Hardware* for details on these configurations.

1.4.2 Hardware for RTOS support

When debugging an RTOS application, using the RTOS extension of RealView Debugger, you can use any processor target supported both by RealView Debugger and by the RTOS support package.

Chapter 2

Tracing with RealView Debugger

This chapter describes how to use RealView® Debugger to set up the conditions for capturing trace information from your target hardware, and how to analyze that trace information using the Analysis window. It contains the following sections:

- *About tracing with RealView Debugger* on page 2-2
- *Getting started* on page 2-8
- *Configuring the ETM* on page 2-16
- *Tracepoints in RealView Debugger* on page 2-29
- *Configuring trace capture* on page 2-35
- *Setting unconditional tracepoints* on page 2-39
- *Setting conditional tracepoints* on page 2-48
- *Managing tracepoints* on page 2-68
- *Capturing trace output* on page 2-72
- *Using the Analysis window* on page 2-73
- *Mapping Analysis window options to CLI commands and qualifiers* on page 2-130
- *Examples of using trace in RealView Debugger* on page 2-134.

2.1 About tracing with RealView Debugger

RealView Debugger enables you to perform tracing on your application or system. You can capture in real-time a historical, non-intrusive trace of instructions and data accesses. Tracing is typically required when a problem results from some interaction between application software and hardware, that occurs while your application is running at full clock speed. These defects can be intermittent, and are difficult to identify through traditional debugging methods that require starting and stopping the processor. Tracing is also useful when trying to identify potential bottlenecks or to improve performance-critical areas of your application.

The following sections introduce the requirements and RealView Debugger features that enable you to capture and analyze trace information:

- *Requirements for tracing*
- *Features for setting up the conditions for trace capture* on page 2-3
- *About Profiling* on page 2-5
- *Tracing with RealView Debugger commands* on page 2-6
- *Available resources* on page 2-6.

2.1.1 Requirements for tracing

RealView Debugger supports tracing with trace hardware. This section describes the system requirements for tracing.

Trace hardware

To capture trace information using trace hardware, you must have the following components:

- A trace solution, which can be either:
 - An *Embedded Trace Macrocell*[™] (ETM) enabled processor that exports trace information to an external trace capture hardware device. See *ARM CPU macrocell with an ETM* on page 2-9 for more details.
 - An ETM-enabled processor that exports trace information directly to an on-chip *Embedded Trace Buffer*[™] (ETB[™]). See *ASIC that supports trace* on page 2-9 for more details.
- A *Joint Test Action Group* (JTAG) interface unit, which must be ARM RealView ICE version 1.1 or higher.

Note

You cannot capture trace information using ARM RealView ICE - Micro Edition (RVI-ME).

For details on connecting your hardware, see Appendix A *Setting up the Trace Hardware*.

For a description of how these hardware components operate together with RealView Debugger to enable you to perform tracing, see *Getting started* on page 2-8.

ETM trace solutions

If you are using an ETM trace solution, you must have the following components:

- an ETM-enabled ARM processor
- an ARM RealView Trace unit.

On-chip trace buffer solution

RealView Debugger trace supports the ARM On-Chip Trace (ETM with ETB) buffer solution.

2.1.2 Features for setting up the conditions for trace capture

The trace conditions you can set depend on the trace features supported by your debug target. Where supported, RealView Debugger enables you to:

- Set up the global trace conditions for generating trace information, including:
 - configuring the ETM (see *Configuring the ETM* on page 2-16)
 - changing the trace buffer size.
- Use an automatic tracing mode to generate trace information (see *Automatic tracing without tracepoints* on page 2-35). By default, only instructions are traced.
- Set specific trace conditions to generate trace information at specific regions of your application. For example, you can:
 - start and stop tracing at specific points
 - specify a range of addresses where tracing of instructions and/or data occurs
 - specify a range of addresses where tracing of instructions and/or data does not occur
 - define a trigger to focus the trace around a region of interest.

For more details on setting specific trace conditions, see *Tracepoints in RealView Debugger* on page 2-29.

You can set these trace conditions on a halted or running core (see *Setting up new trace conditions on a running core* on page 2-15).

———— **Note** —————

If you want to trace data, you must use either:

- an automatic tracing option that includes data
 - a trace range that includes data, even when you use triggers and trace start and end points.
-

2.1.3 About Profiling

In addition to analyzing the trace information directly, you can perform a statistical analysis of your trace information. This enables you to display details such as the amount of time your application spends executing certain functions as a percentage of the overall execution time. You can also display the information graphically as a histogram.

For ETM-enabled hardware, you can obtain profiling information using either or both of the following features (see *Configuring the ETM* on page 2-16 for details on configuring these features):

Timestamping

Analyzing timestamp values enables you to see, for example, when pauses have occurred in processor execution, and how long it takes between successive invocations of a particular section of code.

With timestamping enabled, your trace capture hardware adds a timestamp value to each line of traced information. Trace information is time stamped at a resolution of 10ns on RealView Trace.

———— Note ————

Timestamping is not available when tracing with an on-chip ETB.

Cycle-accurate tracing

When profiling the execution of critical code sequences, it is often useful if you can observe the exact number of cycles that a particular code sequence takes to execute.

Cycle-accurate tracing causes the trace capture hardware to capture trace on all cycles, even if there is no trace to output on that cycle. Therefore, you can determine the number of cycles taken by a region of code by counting the number of cycles of traced capture.

———— Note ————

Cycle-accurate tracing is disabled if the processor enters debug state.

———— Note ————

If you have an ETMv3 core, then instruction count-based profiling information is shown, unless you enable one of these options.

RealView Debugger displays profiling information in the Profile tab of the Analysis window (see *About the Analysis window* on page 2-11).

2.1.4 Tracing with RealView Debugger commands

You can use the RealView Debugger *Command-Line Interface* (CLI) to set up the conditions for capturing trace information and to analyze the captured trace information. You can also include the commands in scripts.

This chapter shows the commands to use to capture and analyze the trace information, and how the commands relate to the equivalent GUI features. Many of the GUI operations output the equivalent CLI commands to the Output pane.

Note

There are some operations, such as profiling, that you cannot perform at the CLI.

For details, see the *RealView Developer Kit v2.2 Command Line Reference Guide*.

2.1.5 Available resources

This section describes the resources that are available for tracing. It contains a section on *Trace hardware with an ETM-enabled ARM processor*.

For information on setting up, see Appendix A *Setting up the Trace Hardware*, and Appendix B *Setting up the Trace Software*.

Trace hardware with an ETM-enabled ARM processor

ETM resources that are relevant to trace capture, such as data comparators, are predetermined by the ETM and can vary with different configurations. The number and size of resources depend on the size of the ETM you are using. These factors determine what trace capture resources you can use when setting tracepoints in RealView Debugger.

The number of resources and the size of the on-chip *First-In-First-Out* (FIFO) buffer are set by the ASIC designer. In ETM7™ and ETM9™, four standard configurations are implemented, each one offering a different trade-off between silicon area, pin count, and complexity of debug features. These configurations are known as Small, Medium, Mediumplus, and Large. ETMs later than ETM9 implement the Mediumplus configuration. For more details about these configurations, see the *Technical Reference Manual* (TRM) for your ETM version and the *Embedded Trace Macrocell Architecture Specification*, which you can obtain from the ARM website.

RealView Debugger interrogates your device, and provides access to only the resources that have been implemented.

Note

RealView Debugger cannot detect whether some optional features of the ETM, such as FIFOFULL, have been connected within the device containing the ETM. To determine this, see the documentation that accompanies your device.

ETM data port width

The width of the ETM data port is configured by the processor manufacturer to one of the following sizes:

- | | |
|---------------|---|
| 4-bit | A 4-bit ETM data port using 9 output signals on the device being traced. |
| 8-bit | An 8-bit ETM data port using 13 output signals on the device being traced. |
| 16-bit | A 16-bit ETM data port using 21 output signals on the device being traced. |
| 24-bit | A 24-bit ETMv3 data port.
This size is not currently available for any trace hardware. |
| 32-bit | A 32-bit ETMv3 data port. This size is currently available only for ETB11™ trace hardware. Also, if you are using ETB11 trace hardware, then this is the only size available. |

2.2 Getting started

This section gives an overview of how trace hardware components operate together to enable you to perform tracing with RealView Debugger, and describes the general procedure for performing tracing on your application after you have configured your system. It contains the following sections:

- *Using the examples*
- *About the trace hardware setup* on page 2-9
- *About the Analysis window* on page 2-11
- *RealView Debugger tracing procedure* on page 2-13
- *Automatically connecting to an analyzer on target connection* on page 2-14
- *Setting up new trace conditions on a running core* on page 2-15.

2.2.1 Using the examples

As an introduction to the RealView Debugger tracing and profiling features, see *Examples of using trace in RealView Debugger* on page 2-134. These examples assume that you are familiar with RealView Debugger, and demonstrate the benefits of using trace and profiling to solve typical development problems.

Preparations for tracing

Before you can use the trace extension to RealView Debugger, you must:

1. Ensure that you have the required system components to perform tracing (see *Requirements for tracing* on page 2-2).
2. Ensure that RealView Debugger is properly connected to your target. You must first ensure that the components are configured properly and connected. See Appendix A *Setting up the Trace Hardware* for details on how to set up and configure your trace hardware. You must also be sure to configure and connect to your chosen target.

2.2.2 About the trace hardware setup

The hardware elements that provide the trace capability are described in the following sections:

- *ASIC that supports trace*
- *Trace capture hardware on page 2-10*
- *JTAG interface unit on page 2-11.*

ASIC that supports trace

You must have a target system that includes an ARM architecture-based ASIC. These configurations are described in the following sections:

- *ARM CPU macrocell with an ETM*
- *ARM processor with ETM and on-chip trace buffer on page 2-10.*

ARM CPU macrocell with an ETM

This is an ARM family CPU that contains EmbeddedICE® logic, and is ETM-enabled. Figure 2-1 shows an example of how RealView Debugger can be joined with ARM ETM-enabled hardware to enable tracing capabilities.

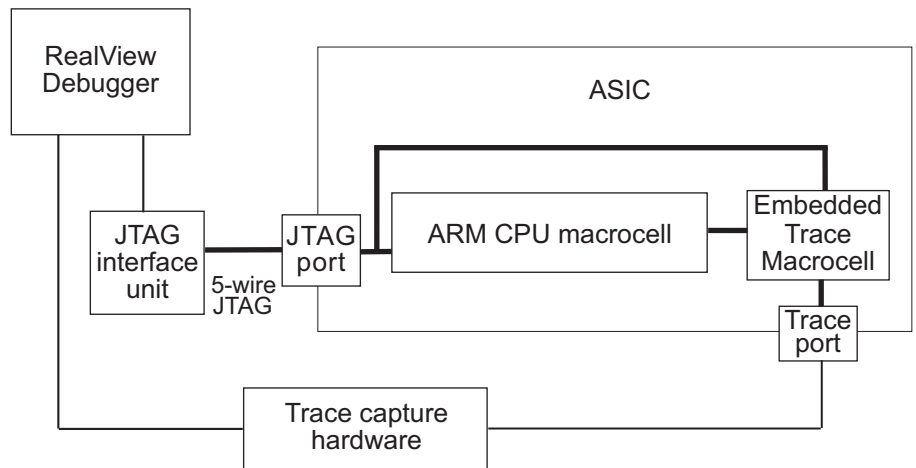


Figure 2-1 Trace hardware setup with trace capture hardware

The ETM monitors the ARM core buses, and passes compressed information through the trace port to the trace capture hardware. To detect sequences of events performed by the processor, the on-chip ETM contains a number of resources that are selected when the ASIC is designed. These resources comprise the trigger and filter logic you utilize within RealView Debugger.

For details on the ETM-enabled ARM processors that are supported by RealView Debugger, see *Requirements for tracing* on page 2-2.

ARM processor with ETM and on-chip trace buffer

This is an ARM family CPU that contains EmbeddedICE logic, and is ETM-enabled with the addition of an ETB. Figure 2-2 shows an example of how RealView Debugger can be joined with ARM ETB-enabled hardware to enable tracing capabilities.

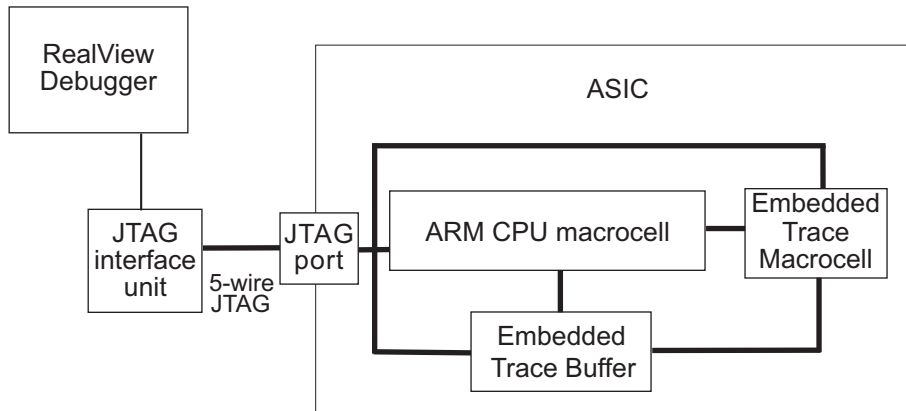


Figure 2-2 Trace hardware setup with ETB

The ETB stores data produced by the ETM. This is essential for processors that run at higher speeds than the trace information can be read by a debugger. Therefore, trace information can be captured in real-time, and accessed by RealView Debugger at a reduced clock rate.

For details on how a processor with an ETB interacts with other trace hardware elements, see the documentation that accompanies the processor.

———— Note ————

Tracing an ASIC that has an ETB does not require any trace capture hardware such as RealView Trace.

Trace capture hardware

This is an external device that stores the information from the trace port. For details on the trace capture hardware devices that are supported by RealView Debugger, see *Requirements for tracing* on page 2-2.

Note

External trace capture hardware is not required for on-chip trace.

JTAG interface unit

This component is a protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the EmbeddedICE logic and the ETM. For details on the JTAG interface units that are supported by RealView Debugger, see *Requirements for tracing* on page 2-2.

2.2.3 About the Analysis window

RealView Debugger provides an Analysis window where you can set up the global trace conditions, including the ETM where supported. The traced information that is captured is placed in a trace buffer, and RealView Debugger presents the trace information in the the Analysis window in a user-readable form. You can then use the Analysis window to:

- view trace data, source, and profiling information using tabbed views
- filter the results of a trace capture
- search for a specific item of trace information
- manipulate the display of trace information.

For detailed instructions on how to use the Analysis window, see *Using the Analysis window* on page 2-73.

Displaying the Analysis window

To display the Analysis window, shown in Figure 2-3, select **View** → **Analysis Window** from the Code window main menu.

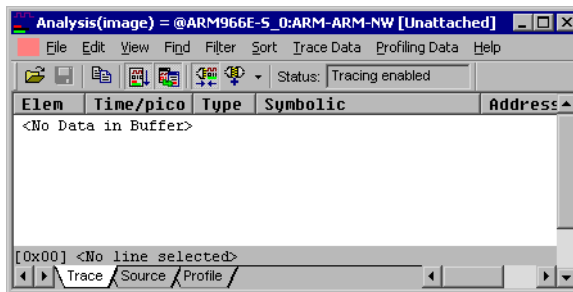


Figure 2-3 Analysis window

Equivalent CLI commands for various Analysis window features

Many of the features in the Analysis window have CLI command equivalents (see *Mapping Analysis window options to CLI commands and qualifiers* on page 2-130). For more details, see the description of the following commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*:

- ANALYZER, to configure the trace logic analyzer
- ETM_CONFIG, to configure the ETM
- TRACEBUFFER, to save, load, search, and filter the trace data.

———— Note ————

There are no equivalent CLI commands for profiling, displaying interleaved source, and displaying inferred registers.

2.2.4 RealView Debugger tracing procedure

It is recommended that you familiarize yourself with the tracing procedure before trying to perform trace capture on your own application using RealView Debugger. It is also suggested that you see *Examples of using trace in RealView Debugger* on page 2-134 to see how trace can be used in a typical development environment. The complete procedure for tracing your application is as follows:

1. Ensure that your target supports trace (see *Requirements for tracing* on page 2-2 for a list of supported targets).
2. Set up your trace hardware (see Appendix A *Setting up the Trace Hardware*).
3. Start RealView Debugger and ensure that your trace software is properly configured (see Appendix B *Setting up the Trace Software*).
4. Connect to the target.
5. If the analyzer is not automatically connected when you connect to the target, select **Tools** → **Analyzer/Trace Control** → **Connect Analyzer/Analysis...** from the Code window main menu.

If you want to automatically connect to an analyzer when you connect to a target, see *Automatically connecting to an analyzer on target connection* on page 2-14.

6. Load the image you want to trace. For details on loading an image, see the chapter *Working with Images* in the *RealView Developer Kit v2.2 Debugger User Guide*.
7. Skip steps 8 and 9 if you want to use the default tracing configuration (see *Automatic tracing without tracepoints* on page 2-35).
8. If you want to configure specific tracing features, change the general tracing configuration options as follows:
 - a. For ETM-enabled targets, use the Configure ETM dialog box. See *Configuring the ETM* on page 2-16 for complete details on using this dialog box.
 - b. For any target, use the options in the **Edit** menu of the Analysis window. See *Configuring trace options* on page 2-103 for details.

———— **Note** ————

The available menu options depends on your target.

—————

9. If you want to limit trace capture to a specific area of your image, determine the area of interest in your source file for which you want to perform tracing, and set tracepoints accordingly. See *Tracepoints in RealView Debugger* on page 2-29 for an introduction to using tracepoints in RealView Debugger.

Note

If you do not set any tracepoints, automatic tracing occurs (see *Automatic tracing without tracepoints* on page 2-35).

10. You can now run your application to capture trace information. The results of the trace capture are returned to the Analysis window (see *About the Analysis window* on page 2-11), where you can analyze the results.

To display the Analysis window, select **View** → **Analysis window** from the Code window main menu.

For detailed information on using this window, see *Using the Analysis window* on page 2-73.

Note

The availability of tracing resources and options in the Analysis window depend on your system configuration. See *Available resources* on page 2-6 for a description of which resources are available with each configuration.

2.2.5 Automatically connecting to an analyzer on target connection

If you have an ETM-based target, you can configure RealView Debugger to automatically connect to an analyzer when you connect to that target. To do this:

1. In the Connection Control window, right-click on the required connection and select **Connection Properties...** from the context menu. The Connection Properties window is displayed, and the CONNECTION= group for the connection is selected.
2. In the left pane, expand the following groups for the connection:
 - a. CONNECTION=
 - b. Advanced_Information
 - c. Default.
3. Select Logic_Analyzer in the left pane.
4. Right click on the Vendor setting in the right pane.
5. Select the manufacturer for your analyzer, for example **ARM**.
6. Select **File** → **Save and Close** from the menu.

When you connect to a target that supports tracing on this connection, RealView Debugger automatically connects the analyzer.

2.2.6 Setting up new trace conditions on a running core

If you have a core that is already running, you can set up new trace conditions on that core without having to stop it. When you attempt to configure new trace conditions on a running core, RealView Debugger displays one of the following prompts:

- When setting or clearing tracepoints, RealView Debugger displays the prompt shown in Figure 2-4.

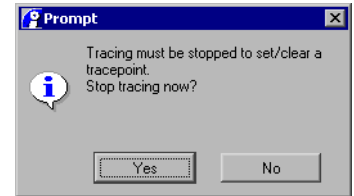


Figure 2-4 Prompt displayed on a running target (tracepoints)

If you click **Yes**, RealView Debugger disables tracing, sets or clears the tracepoint as required, and updates the trace buffer. To enable tracing again, select **Edit → Tracing Enabled** from the Analysis window menu.

- When configuring the global trace conditions, RealView Debugger displays the prompt shown in Figure 2-5.

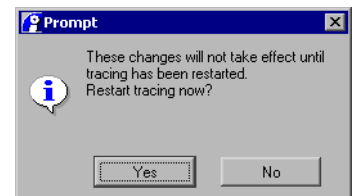


Figure 2-5 Prompt displayed on a running target (global trace conditions)

If you click **Yes**, RealView Debugger temporarily disables tracing, sets up the global trace condition that you requested, and enables tracing again.

See *Configuring trace capture* on page 2-35 for more details on configuring trace conditions.

2.3 Configuring the ETM

Use the Configure ETM dialog box to configure the ETM behavior. The settings you configure in this dialog box apply to any tracing you perform, whatever trace capture criteria you set and whatever the number of trace captures you perform (see *Configuring trace capture* on page 2-35).

Many of the facilities provided by the ETM can be enabled or disabled in the hardware by the manufacturer, and some features used by RealView Debugger depend on the support provided by the trace capture hardware you are using. RealView Debugger provides mechanisms to determine which facilities are actually implemented by the ETM and the trace capture hardware.

RealView Debugger enables or disables ETM configuration settings depending on whether they are available on your target. In cases where a value, such as the Trace port mode, is fixed because of a target-specific restriction, the value is displayed in the Configure ETM dialog box, but the setting to amend it is disabled. However, there are facilities that might not be detected, and there are some optional features that cannot be autodetected. For example, if the FIFOFULL logic is present but not connected to the processor, the ETM does report that FIFOFULL logic is present, but the FIFOFULL logic does not operate.

This section includes:

- *Displaying the Configure ETM dialog box*
- *Configure ETM dialog box interface components* on page 2-19
- *Equivalent CLI command qualifiers* on page 2-27.

2.3.1 Displaying the Configure ETM dialog box

To display the Configure ETM dialog box, select either of the following:

- **Edit → Configure Analyzer Properties...** in the Analysis window.
- **Tools → Analyzer/Trace Control → Configure Analyzer Properties...** from a Code window.

After you have configured the ETM, click **OK** to save the configuration, or **Cancel** to discard any changes.

Configure ETM dialog box for non-v3 architectures

The appearance of the Configure ETM dialog box varies depending on the ETM architecture you are using. For architectures other than ETM v3, the Configure ETM dialog box appears as shown in Figure 2-6. The ETM architecture and protocol versions are displayed at the top of the Configure ETM dialog box. See *Configure ETM dialog box interface components* on page 2-19 for details of the settings on this dialog box.

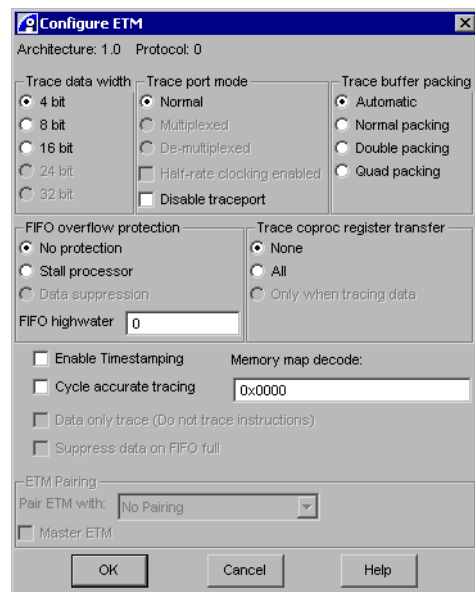


Figure 2-6 The Configure ETM dialog box for non-v3 architectures

Configure ETM dialog box for ETM v3 architectures

For ETM v3, the Configure ETM appears as shown in Figure 2-7. See *Configure ETM dialog box interface components* on page 2-19 for details of the settings on this dialog box.

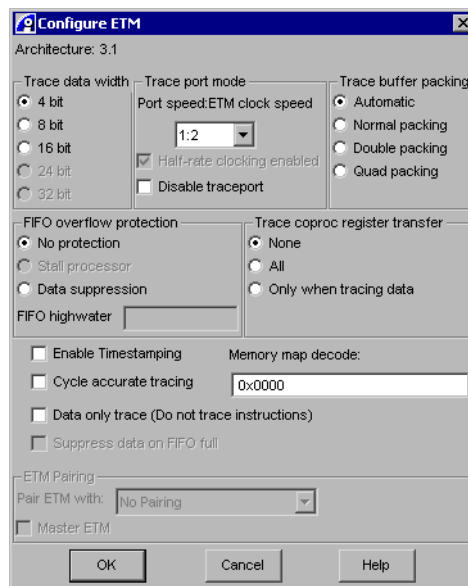


Figure 2-7 The Configure ETM dialog box for ETMv3

2.3.2 Configure ETM dialog box interface components

The interface components in the Configure ETM dialog box are described in the following sections:

- *Trace data width*
- *Trace port mode* on page 2-20
- *Trace buffer packing* on page 2-21
- *FIFO overflow protection* on page 2-22
- *Trace coproc register transfer* on page 2-23
- *Enable Timestamping* on page 2-24
- *Cycle accurate tracing* on page 2-24
- *Data only trace (Do not trace instructions)* on page 2-25
- *Suppress data on FIFO full* on page 2-25
- *Memory map decode* on page 2-26.

Note

Some of these settings might be grayed out, depending on your system design or current configuration, for example, packing modes determine what port widths are available.

Trace data width

This control enables you to set the width of the ETM trace data port, which determines the number of trace port pins that are used to broadcast trace information (see *ETM data port width* on page 2-7). This can be useful, for example, when trace port pins are multiplexed onto *General Purpose Input/Output* (GPIO) pins, and the hardware is configured to use these pins in their GPIO role. Only those widths supported by your trace capture hardware and target device are enabled for selection.

Note

For profiling, a smaller trace port width is more useful. This is because profiling does not require as fine a granularity of trace information, and gives the best use of your trace buffer.

Trace port mode

This control specifies the way the trace port operates:

- For architectures other than ETMv3, select one of the following values:

Normal	The normal mode. Trace data from the ETM is written to the output pins at the processor frequency.
Multiplexed	Use this to reduce the number of output pins used by the trace port. Two output signals are output on the same pin by clocking the signals at double the normal rate.
De-Multiplexed	Use this to reduce the signal switching frequency of the trace port signals. One output signal is output on two pins, so the pins are clocked at half the normal rate.

Note

- Only **Normal** mode operation is possible when you are using ETM hardware implementing ETM Architecture version 1.1 or below.
 - If you use multiplexed or demultiplexed clocks, you might have to alter the configuration of your trace capture hardware.
-

- For ETMv3 architectures, select the **Port speed:ETM clock speed** ratio from the drop-down list. This enables the trace port to run at a different speed to that of the core. Therefore, you can capture trace from cores that run faster than the trace port capture speed. Available options are:
 - **Dynamic** (for use with on-chip trace buffers). This is the only option available if you are using ETB11.
 - **1:1** (the port speed is the same as the ETM clock speed).
 - **1:2** (the port speed is half the ETM clock speed).
 - **1:3** (the port speed is one third of the ETM clock speed).
 - **1:4** (the port speed is one quarter of the ETM clock speed).
 - **2:1** (the port speed is double the ETM clock speed).
 - **Implementation defined** (defined by the ASIC manufacturer).

Note

The **Port speed:ETM clock speed** ratio defines the rate of the trace port, not the trace clock speed. The trace clock speed is always half the trace port rate, because half-rate clocking is mandated by the ETMv3 architecture.

You can also use this section of the dialog box to enable or disable half-rate clocking (only for ETMv1 and ETMv2 architectures), or suppress output from the trace port:

Half-rate clocking enabled

Select **Half-rate clocking enabled** if you want to set the ETM half-rate clocking signal. The effect of this signal is dependent on the implementation of your system. For more details on half-rate clocking, see the *ETM control register* section of the *ARM Embedded Trace Macrocell Specification*.

————— **Note** —————

- This setting is not available when you are using ETM hardware implementing ETMv1.0. Hardware implementing ETMv1.1 might support the setting, but RealView Debugger cannot detect whether it does.
- If you enable half-rate clocking, you might have to alter the configuration of your trace capture hardware.
- This capability is not supported by all trace capture hardware. See the documentation that accompanies your hardware to see if it is available.

Disable traceport

Select **Disable traceport** if you want to suppress the output from the trace port. This is useful if your hardware has two or more ETMs sharing a single trace port.

Trace buffer packing

This setting is specific to the *Trace Port Analyzer* (TPA). It enables you to select the mode in which trace data is packed into the buffer of the TPA. Double and Quad packing modes enable multiple consecutive trace samples to be written to the same memory location within the TPA. This increases the trace depth, but results in coarser timestamping. Half-rate clocking and packing modes are available at **TRACECLK** frequencies above 100MHz.

The trace buffer packing modes available are:

Automatic This mode enables the TPA to select the best packing mode for the current port width. Select this option if timestamping is not enabled.

Normal Packing

In this mode, the TPA records one timestamp for each trace sample. This gives the best possible resolution of timestamps, at the expense of trace depth.

Double Packing

In this mode, the TPA records one timestamp for every two consecutive trace samples. This reduces the resolution of the timestamps, but increases the trace depth.

Quad Packing

In this mode, the TPA records one timestamp for every four consecutive trace samples. This gives the greatest trace depth, but reduces the resolution of timestamps even more.

Note

This setting is only available for ETM targets that are connected using RealView Trace. For RealView Trace, **Double Packing** doubles the trace depth and **Quad Packing** quadruples it. For details of the trace depths supported by your TPA, see the documentation for your TPA.

Some combinations of packing mode and port width might not be valid for your system. For example, **Quad Packing** might only be possible if you are using a narrow port width. See the documentation for your TPA for more details.

FIFO overflow protection

The ETM contains a FIFO buffer that holds the traced data for transmission through the trace port. When this FIFO buffer becomes full, trace information is lost unless you have programmed the ETM to stall (temporarily stop) the processor. The options in this section of the dialog box enable you to protect against FIFO overflows.

Select one of the following options:

No protection

Select this option if you do not want to use FIFO overflow protection.

Stall processor

Select this option if you want the system to stall the processor until the FIFO buffer is empty. You must also set the **FIFO highwater** by typing a value into the text box. When the number of bytes left in the FIFO buffer

is reduced to the number of bytes you set in **FIFO highwater**, the processor stalls as soon as possible. It restarts when the FIFO buffer is empty.

Note

This capability is not supported by all systems. See the documentation that accompanies your system to find out whether FIFOFULL is implemented.

Data suppression

Select this option to instruct RealView Debugger to stop tracing data when the FIFO is close to the overflow limit. This helps to prevent a FIFO overflow, because the bandwidth required for instruction tracing is much lower than that required for data tracing.

The following warning message is displayed when data is being suppressed, and again when data is unsuppressed:

Warning: Data suppression protected ETM FIFO from overflow

Note

This option is only available if you are using ETMv3, and is grayed out for all other ETM architectures.

Trace coproc register transfer

This area of the dialog box enables you to specify when *Coprocessor Register Transfers* (CPRTs) are traced. CPRTs are the instructions *Move Coprocessor from ARM Register* (MCR) and *Move ARM Register from Coprocessor* (MRC).

Select one of the following options:

None Do not trace CPRTs.

All Trace all CPRTs.

Only when tracing data

Only trace CPRTs when data is being traced. To specify whether or not to trace data, select the **Edit → Data tracing mode** option in the Analysis window menu (see *Data Tracing Mode* on page 2-108).

Note

This setting is only available if you are using ETMv3. It is grayed out for all other ETM architectures.

Enable Timestamping

This setting enables the timestamp recording logic in your trace capture hardware (such as RealView Trace). Timestamps are displayed in the **Time/unit** and **+Time** columns of the Analysis window (see *Columns in the Trace tab* on page 2-80). To change the format in which timestamps are displayed, select the **View → Scale Time Units...** option in the Analysis window menu (see *Scale Time Units...* on page 2-111).

If you want to view profiling information, you must enable either timestamping or cycle-accurate tracing, or both (see *About Profiling* on page 2-5). If you have an ETMv3 core, then instruction count-based profiling information is shown, unless you enable one of these options. Better results in profiling are generally obtained with timestamping enabled. However, if trace pauses are unimportant, due to the trace information being time stamped at a resolution of 10ns on RealView Trace, profiling of ETMv3 cores clocked above 100MHz are more precise if you specify cycle-accurate tracing without timestamping. The ETMv3 trace protocol derived core cycles are used as the timing data for profiling. See *Viewing profiling information* on page 2-91 for information on the profile view.

Transmitting the timestamps uses additional bandwidth on the trace capture hardware to host connection. Storing the timestamps in the trace capture hardware might reduce the maximum length of trace that you can capture. Therefore, only enable this feature when you have to.

Note

This feature is not supported when using an on-chip ETB, and by some types of trace capture hardware. See the documentation that accompanies your hardware for more information.

Cycle accurate tracing

This setting determines whether the ETM operates in cycle-accurate mode (see *About Profiling* on page 2-5):

- When **Cycle accurate tracing** is selected, the ETM records the number of cycles executed while tracing is enabled. This includes cycles during which no trace information is normally returned, such as memory wait states.

Note

For ETMv2, and earlier, cycle-accurate tracing causes Wait (**WT**) and Trace Disabled (**TD**) pipeline status signals to be transmitted to your trace capture device. To save space, RealView Trace discards the **TD** signals.

The **Elem** column of the Analysis window (see *Columns in the Trace tab* on page 2-80) shows the cycle number of the cycle in which each instruction was executed. The count does not include cycles executed during a trace discontinuity. You must use the **Time/unit** column, which displays timestamp values, to measure across discontinuities in the trace output.

If you want to view profiling information, you must enable either timestamping or cycle-accurate tracing, or both. If you have an ETMv3 core, then instruction count-based profiling information is shown, unless you enable one of these options. Better results in profiling are generally obtained with timestamping enabled. However, if trace pauses are unimportant, due to the trace information being time stamped at a resolution of 10ns on RealView Trace, profiling of ETMv3 cores clocked above 100MHz are more precise if you specify cycle-accurate tracing without timestamping.

- If **Cycle accurate tracing** is not selected, the ETM does not record cycle counts. The **Elem** column in the Analysis window shows a row number relative to the trigger point, if one has been set.

Note

Cycle-accurate tracing fills up the capture buffer faster than non cycle-accurate tracing because all Wait cycles are captured, even if there is no trace output on a cycle.

Data only trace (Do not trace instructions)

This setting forces the tracing of data transfers only. Otherwise, both data and instructions are traced. In addition, cycle-accurate tracing is disabled, because no cycle values are recorded.

You must also enable data tracing if you select this option. Otherwise, you do not receive any trace, or you might only see discontinuities or status messages in the trace.

Note

This setting is only available if you are using ETMv3, and is grayed out for all other ETM architectures.

Suppress data on FIFO full

This setting suppresses the output from the trace port after a FIFO overflow occurs. Some versions of the ETM produce incorrect trace data following FIFO overflow. This only occurs on cached processors with slow memory systems, and happens when a

cache miss occurs at the same time that the FIFO on the ETM overflows. If you select this setting, the decompressor suppresses the data that the ETM might have traced incorrectly. However, some correctly traced data might also be suppressed.

———— **Note** ————

This setting is disabled if your ETM does not generate incorrect trace data under these circumstances.

Memory map decode

This is an implementation-dependent value that varies depending on the memory map decode logic present in your system. This value is written to a control register, intended to configure the memory map decode hardware. For more details, see your system ASIC documentation.

2.3.3 Equivalent CLI command qualifiers

Table 2-1 shows the Configure ETM dialog box settings and the equivalent qualifiers to use with the ETM_CONFIG command to perform the same function in the CLI.

Table 2-1 ETM Configuration settings to command qualifier mapping

Configure ETM dialog box setting	Command qualifier
Trace data widths:	
• 4 bit	port_width:0
• 8 bit	port_width:1
• 16 bit	port_width:2
• 24 bit	port_width:3
• 32 bit (ETB11 only)	port_width:4
Trace port modes (non-v3 architecture):	
• Normal	nomultiplex
• Multiplexed	multiplex
• Demultiplexed	demultiplex
Trace port modes (ETMv3 only), of the form <i>Port_speed:ETM_clock_speed:</i>	
• 1:1	portratio:0
• 1:2	portratio:1
• 1:3	portratio:2
• 1:4	portratio:3
• 2:1	portratio:4
• Use dynamic ratio modes for on-chip trace.	portratio:5
• Use the implementation-defined mode, if implemented by the ASIC designer.	portratio:6
Half-rate clocking enabled	half_rate
Disable traceport	disableport
Trace buffer packing:	
• Automatic	packauto
• Normal packing	packnormal
• Double packing	packdouble
• Quad packing	packquad

Table 2-1 ETM Configuration settings to command qualifier mapping (continued)

Configure ETM dialog box setting	Command qualifier
FIFO overflow protection:	
• Stall processor	stall_full
• Data suppression	suppressdata
• FIFO highwater	FIFO_hw: <i>n</i>
Trace coproc register transfer:	
• Stall processor	coprocessor
• Only when tracing data	filtercoprocessor
Enable Timestamping	time_stamps
Cycle-accurate tracing	cycle_accurate
Data only trace (Do not trace instructions)	dataonly
Suppress data on FIFO full	datasuppression
Memory map decode	mmap_decode: <i>n</i>

2.4 Tracepoints in RealView Debugger

Tracepoints enable you to set conditions for generating trace information. This section gives an overview of the tracepoints available in RealView Debugger. It includes:

- *Unconditional and conditional tracepoints*
- *Tracepoint types*
- *Setting trace ranges in conjunction with trace start and end points* on page 2-33.

You can set tracepoints in the Code window, either through dialog boxes, or by issuing CLI commands directly or in scripts.

2.4.1 Unconditional and conditional tracepoints

Tracepoints can be:

Unconditional	These include individual trigger points, trace start and end points, and trace ranges for instruction and data accesses. See <i>Setting unconditional tracepoints</i> on page 2-39 for details.
Conditional	These include AND or OR conditions, tests on the number of executions, and complex comparisons. See <i>Setting conditional tracepoints</i> on page 2-48 for details.

2.4.2 Tracepoint types

The types of tracepoint available are described in the following sections:

- *Trigger*
- *Trace start point and trace end point* on page 2-30
- *Trace range* on page 2-31
- *ExternalOut Points* on page 2-32.

Trigger

A trigger enables you to focus trace collection around a specific region of interest. Although you can set multiple trigger conditions, only the first trigger condition that is hit is used in the trace buffer.

Only instructions are traced when you use a trigger. If you want to trace data, then you must also set a trace range that includes data (see *Trace range* on page 2-31).

When a trigger is activated, the ETM outputs a trigger event, either over the trace port to your trace capture hardware (see Figure 2-1 on page 2-9) or to an on-chip ETB (see Figure 2-2 on page 2-10). The trigger event also indicates the region around the trigger where trace information is to be collected (see *Trigger Mode* on page 2-107):


Trace before This indicates that only trace information before the trigger is to be collected. This is used to find out what has caused a certain event, for example, to see what sequence of code was executed before entering an error handler routine. A small amount of trace data is often also collected after the trigger condition.

Trace around

This indicates that trace information is to be collected before and after the trigger. You can alter the amount of trace information collected before and after the trigger by using a trace start point to enable tracing before the trigger, and a trace end point to disable tracing after the trigger.

Trace after This indicates that trace information must be collected from the trigger point onwards. It is used to find out what happens after a particular event, for example what happens after entering an interrupt service routine. A small amount of trace data is often also collected before the trigger condition.

You can also choose to stop the processor when the trigger event occurs. However, the trace information that is collected when tracing around or after the trigger depends on your trace capture hardware. For RealView Trace, trace information is captured before the trigger, but none after the trigger.

A trigger is identified by an arrow  in the left margin, next to the line of code you select.

You can set triggers over a range of addresses, where any instruction in that range activates the trigger. However, only the first instruction that is hit within the range activates the trigger. In this case, the range is indicated in the same way as any other trace range (see *Trace range* on page 2-31).

Trace start point and trace end point

Set a trace start point and a trace end point to indicate where tracing is to be enabled and disabled. The advantage of enabling and disabling the capture of trace information at specific parts of your image is that it effectively increases the amount of useful information that can be captured for a given size of trace buffer. Therefore, you can enable selective tracing over a longer time period.

Only instructions are traced between trace start and end points. If you want to trace data, then you must also set a trace range that includes data (see *Trace range*), and that also overlaps the region bounded by the trace start and end points (see *Setting trace ranges in conjunction with trace start and end points* on page 2-33 for more details).

The support for trace start and end points depends on your version of ETM:

- In ETMv1.1 or earlier, RealView Debugger uses the ETM state machine, if available, to support a limited number of start and end points, either:
 - up to four start points and up to two stop points
 - up to two start points and up to four stop points.



The limitation option used by RealView Debugger depends on the type of tracepoints you want to set. For example, if you are using ETMv1.1 or earlier, and set three trace start points, then the first case is assumed. That is, you can set up to four start points, but are limited to two stop points.

- In ETMv1.2 or later, you can turn instruction tracing on or off whenever certain instructions are executed or when specified data addresses are accessed. This means that you can trace functions, subroutines, or individual variables held in memory.

You can enable or disable tracing when any single address comparator matches. The effect is only precise if the address comparator is instruction execution or data address based. You can use instruction fetch comparisons, but the effect is not precise.



Trace information is returned from the specified start point until the specified end point, including any areas that are branched to. However:

- if you do not set a trace end point, trace information is returned from the trace start point until execution stops
- if you do not set a trace start point, then trace information is returned from the location currently held in the PC until the trace end point is reached.

The trace start and end points are identified respectively by the arrows  and  in the left margin. You must set each point individually at the required lines of code.

Trace range

A trace range indicates an area of your image where a specific type of trace information is captured (see *Include trace range* on page 2-32) or is not captured (see *Exclude trace range* on page 2-32).

The start and end positions of the trace range are identified respectively by the arrows  and  in the left margin. You can set each position individually at the required lines of code, or select a range of lines and set the trace range with a single operation (see *Setting a trace range for selected source lines or disassembly* on page 2-46).

If you do not set an end position for the range, the default, 0xFFFFFFFF, is used.

You can use trace ranges in conjunction with trace start and end points (see *Setting trace ranges in conjunction with trace start and end points* on page 2-33 for more details).

Include trace range

An Include trace range indicates an area of your image where you want to capture a specific type of trace information. Information is captured only for the specified area, and not for any areas that are branched to. These branches are represented as Trace Pause status lines in the Analysis window (see *Status lines in the Trace tab* on page 2-86). You can set the following types of Include trace range:

- instructions only
- instructions and data.

Exclude trace range

An Exclude trace range indicates an area of your image where you do not want to capture trace information. You can set the following types of Exclude trace range:


- instructions and data (cannot be used in conjunction with Include trace ranges)
- data only (can be used in conjunction with Include trace ranges).

———— Note ————

Because of limitations in the ETM hardware, you can set both Include and Exclude trace ranges only in specific circumstances. If you attempt to use an Exclude trace range incorrectly, RealView Debugger displays an error in the Output pane.

ExternalOut Points

Each ExternalOut point controls a single-bit output signal from the ETM. Up to four signals are available, ExternalOut1, ExternalOut2, ExternalOut3, and ExternalOut4. These can be used to enable or disable trace capture, or to enable or disable a device or peripheral. The ASIC manufacturer determines the availability and use of these output signals. See your ASIC documentation for details.

An ExternalOut point that is set on a single line of code is identified by an arrow  in the left margin, next to the line of code you select.

You can also set ExternalOut points over a range of addresses. In this case, the range is indicated in the same way as any other trace range (see *Trace range* on page 2-31).

2.4.3 Setting trace ranges in conjunction with trace start and end points

On hardware targets, you can set a trace range in conjunction with trace start and end points. If you do, then the position of the trace start and end points determines the region where trace is captured:

- If the trace range is within the region defined by the trace start and end points then trace information is captured for the whole range.
- If the trace start point is after the start of the range, no trace information is captured between the start of the range and the trace start point.
- If the trace end point is before the end of the range, no trace information is captured between the trace end point and the end of the range.

To summarize, trace start and end points determine where tracing is enabled, but the trace ranges determine what information is captured. In the example shown in Figure 2-8 on page 2-34, trace information is captured only between addresses 0x000085E4 and 0x0000861C.

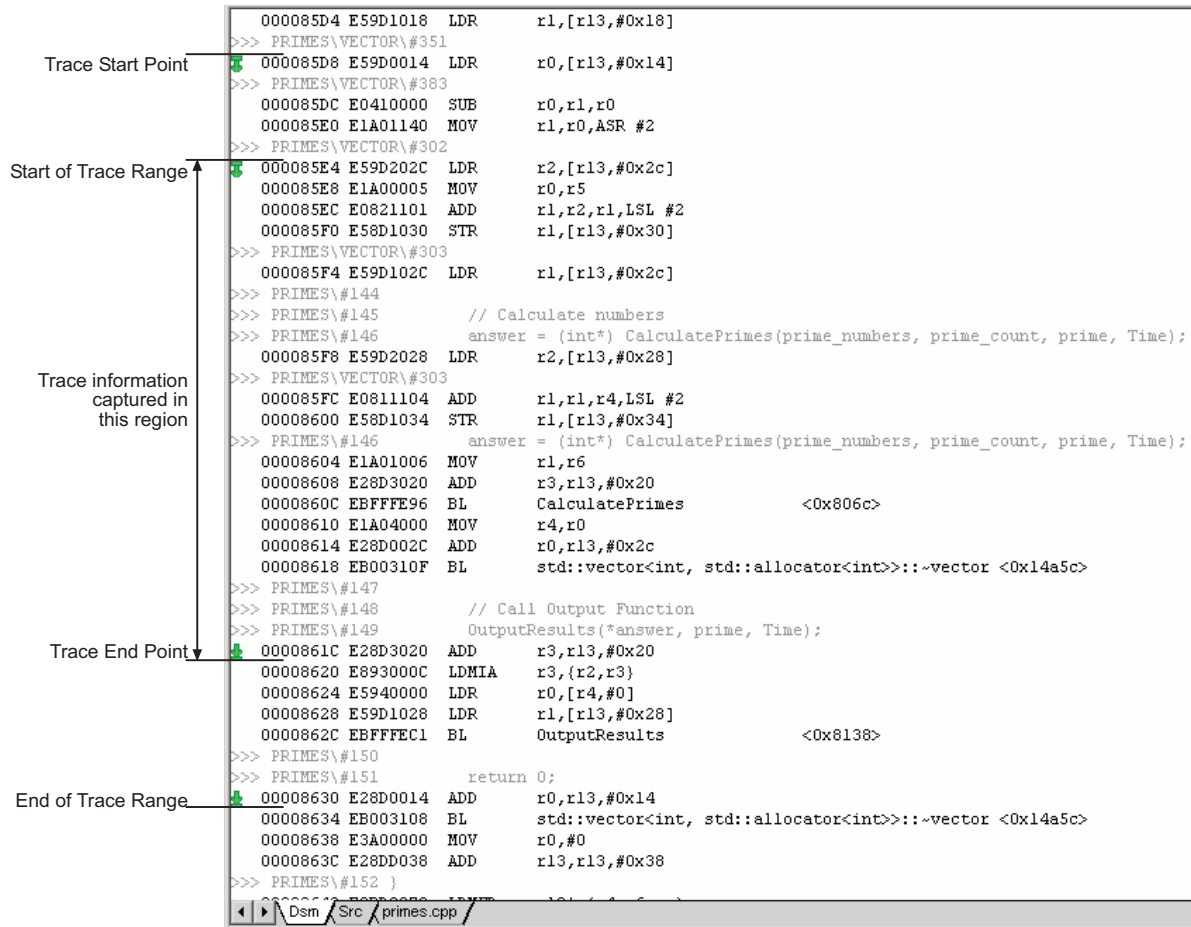


Figure 2-8 Trace start and end points with a trace range

2.5 Configuring trace capture

This section describes how to configure the conditions for trace capture. It contains the following sections:

- *Setting up the global conditions for trace capture*
- *Automatic tracing without tracepoints*
- *About tracing with tracepoints* on page 2-36
- *Setting tracepoints* on page 2-37
- *Setting tracepoints on exception vectors* on page 2-38.

2.5.1 Setting up the global conditions for trace capture

You can set global conditions for trace capture, which depend on your debug target. You can access these configuration settings from the **Edit** menu of the Analysis window (see *Configuring trace options* on page 2-103). You can:

- Change the size of the trace buffer.
- Choose how trace information is to be captured around a trigger tracepoint.
- What trace information to capture when no tracepoints are set. See *Automatic tracing without tracepoints* for details.
- Choose to trace data only (ETMv3 only). You must set tracepoints with this option.
- Choose to stop the processor when a trigger occurs.
- Specify what information to trace for data transfer instructions.
- Configure the ETM.

See *Configuring the ETM* on page 2-16 for full details on how to configure the ETM.

Other options are available if your target supports them.

2.5.2 Automatic tracing without tracepoints

If you do not set any tracepoints, then RealView Debugger automatically traces the whole image. What is traced is determined by the Automatic Tracing Mode. The supported modes depend on your target:

- Instructions only (the default)
- Data only (ETMv3 only)
- Instructions and data.

For more details about setting the Automatic Tracing Mode, see *Automatic Tracing Mode* on page 2-108.

Note

If you turn off Automatic Tracing Mode, you must use tracepoints to capture trace information.

2.5.3 About tracing with tracepoints

If you want to limit tracing to specific areas of your image, then RealView Debugger enables you to set a variety of tracepoints (see *Tracepoints in RealView Debugger* on page 2-29). Therefore, you can control the amount and content of application information that is traced. You can set a tracepoint on any of the following:

- a line or range of source code
- a line or range of assembly code
- a function name
- a function entry point
- a memory address or range of memory addresses.

After you have configured the details for trace capture and executed your application, the captured information is returned to the Analysis window. From there, you can perform additional filtering by narrowing the results of the capture (see *Filtering captured information* on page 2-118).

2.5.4 Setting tracepoints

To set unconditional or conditional tracepoints, you must:

1. Ensure that you are connected to your target, and that the trace analyzer is connected.
2. Ensure that your image is loaded into RealView Debugger.
3. Determine the area of interest within your application for which you want to collect trace information. Depending on the area of interest, you must give focus to your application within one of the following RealView Debugger Code window views:

File Editor pane Src tab

If you want to set tracepoints for specific lines of code in your source files. You might want to turn on line numbering to make this easier.

File Editor pane Dsm tab

If you want to set tracepoints at specific addresses or address ranges in your image.

Memory pane

If you want to set tracepoints at specific memory addresses or address ranges in your image.

4. Choose from the following types of trace-capture setting types available, depending on the complexity of the criteria you want to specify for the capture:

Unconditional tracepoints

For setting trigger points, trace start and end points, or trace ranges for memory and data accesses. See *Setting unconditional tracepoints* on page 2-39 for details.

Conditional tracepoints

For setting AND or OR conditions, counter conditions, and complex comparisons. These conditions can involve any supported combination of trigger points and ranges. See *Setting conditional tracepoints* on page 2-48 for details.

5. Select **View** → **Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane. This pane enables you to view, edit and track tracepoints. It also shows you the CLI command that was used either by you or RealView Debugger to set the tracepoint. See *Managing tracepoints* on page 2-68 for more information on this pane.

You must now run your image to begin trace capture with the details you have set, as described in *Capturing trace output* on page 2-72. Also, see the chapter that describes controlling execution in the *RealView Developer Kit v2.2 Debugger User Guide* for detailed instructions about running images in RealView Debugger.

Note

The availability of tracepoint options depends on the resources you have available and the type of connection you are using. If you have already used a number of resources, some of the options described in this section might not all be available to you.

2.5.5 Setting tracepoints on exception vectors

If you want to set a tracepoint on an exception vector, such as Data Abort, remember to disable the vector catch for that exception before running the image. This prevents the exception being caught, and the exception can be recorded in the trace buffer. The text recorded in the trace buffer for each exception is shown in Table 2-2. See the example described in *Finding the cause of a data abort* on page 2-135 for instructions on how to trace a Data Abort exception.

Table 2-2 Text recorded in trace buffer for exceptions

Exception	Text recorded
Reset	RESET
Undefined instruction	UNDEF EXCEPTION
Software Interrupt (SWI)	Branch to <SWI>
Prefetch Abort (instruction memory read abort)	PREFETCH ABORT
Data Abort (data memory read or write abort)	DATA ABORT
IRQ (normal interrupt)	INTERRUPTED
FIR (fast interrupt)	FAST INTERRUPT
Secure Monitor Interrupt (SMI) ^a	SECURE MONITOR INTERRUPT

a. This is supported only on ETMv3.2, or later.

2.6 Setting unconditional tracepoints

This section describes how to set unconditional tracepoints (see *Tracepoints in RealView Debugger* on page 2-29). You can set unconditional tracepoints using one of the following methods:

- use the Set/Toggle Tracepoint List Selection dialog box (see *Using the Set/Toggle Tracepoint List Selection dialog box*)
- set a trace range on selected lines of source or disassembly in the File Editor pane (see *Setting a trace range for selected source lines or disassembly* on page 2-46)
- manually with the Set/Edit Tracepoint dialog box (see *Setting unconditional tracepoints with the Set/Edit Tracepoint dialog box* on page 2-46)
- use the any of the trace commands, TRACE, TRACEDATAACCESS, TRACEDATAREAD, TRACEDATAWRITE, TRACEEXTCOND, TRACEINSTREXEC, or TRACEINSTRFETCH (see the description of these commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

When you set a tracepoint, such as a trigger, a corresponding **Clear** option becomes available in the List Selection dialog box. You can select the **Clear** option to remove the tracepoint you have set, and the arrow in the left margin of your code is removed. The option **Clear Range** removes both the start and end points of the range you have set.

———— Note ————

You can set multiple tracepoints on an individual source line. Therefore, if you clear one tracepoint and another exists at the same location, the arrow icon that indicates the tracepoint is still present.

2.6.1 Capturing instructions and data

If you want to capture instructions and data, for ETM-enabled processors, set an unconditional trace range with the Set/Toggle Tracepoint List Selection dialog box, using the **Start of Trace Range (Instruction and Data)** and **End of Trace Range (Instruction and Data)** options (see *Using the Set/Toggle Tracepoint List Selection dialog box*).

2.6.2 Using the Set/Toggle Tracepoint List Selection dialog box

The Set/Toggle Tracepoint List Selection dialog box, shown in Figure 2-9 on page 2-40, enables you to set trace ranges, trace start and end points, and triggers. These can be used individually or in conjunction with other tracepoints, to ensure capture of trace information for the precise area of interest.

Note

The tracepoint type you select from this dialog box is set only at the chosen line of code or address. Therefore, to specify a trace range with this dialog box, you must set the start and end points individually. You must use other methods if you want to set a trace range in a single operation (see *Setting a trace range for selected source lines or disassembly* on page 2-46).

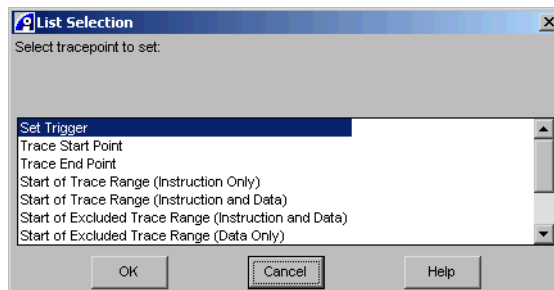


Figure 2-9 List Selection dialog box for setting unconditional tracepoints

To set an unconditional tracepoint using the Set/Toggle Tracepoint List Selection dialog box:

1. Display the dialog box, shown in Figure 2-9, using one of the following methods:
 - In the File Editor pane, right-click on the gray bar to the left of the source and select the **Set/Toggle Tracepoint...** option from the context menu.
 - In the Memory pane, right-click on a memory location and select the **Set Tracepoint...** option from the context menu (*Using the Set/Toggle Tracepoint List Selection dialog box* on page 2-39).
 - Enter the following CLI command in the **Cmd** tab of the Output pane:
`TRACE, prompt address`
 The *address* can be an instruction address or a line reference in the source code, for example, `\MODULE\#100:1`. For more details on the TRACE command, see the *RealView Developer Kit v2.2 Command Line Reference Guide*.

Note

The options that appear in this dialog box depend on the available resources and on the target that you are using. In some cases, you must clear an existing tracepoint or range to free up the resources you might require for a new tracepoint or range.

2. Select the required tracepoint type from the list. The options are described in the following sections:
 - *Setting a trigger*
 - *Setting a trace start and end point* on page 2-42
 - *Setting a trace range* on page 2-42
 - *Setting an ExternalOut Point* on page 2-44.
3. Click **OK** to set the tracepoint and close the dialog box.

For hardware targets, you can set trace ranges in conjunction with trace start and end points. See *Setting trace ranges in conjunction with trace start and end points* on page 2-33 for a description of the tracing behavior in this situation.

For ETM-based traces, when you set unconditional tracepoints with this dialog box, RealView Debugger generates equivalent CLI commands. The basic CLI command that is used by RealView Debugger to set the tracepoint types from this dialog box is:

```
TRACEINSTRExec, hw_out="Tracepoint Type=type" address | address_range
```

The command is modified depending on the type of tracepoint you select (see the description for each tracepoint type). See *Equivalent CLI commands used for unconditional tracepoints* on page 2-45 for details of the CLI commands that are generated for each tracepoint type.

————— **Note** —————

The amount of trace information returned depends on the buffer size that is currently set (see *Configuring trace options* on page 2-103).

Setting a trigger


To set a trigger (see *Trigger* on page 2-29), select the **Set Trigger** option in the Set/Toggle Tracepoint List Selection dialog box. An arrow ➡ is placed in the left margin next to the line of code you have selected.

Use this in conjunction with the **Trigger Mode** options on the Analysis window to determine whether trace information is collected before, around or after the trigger (see *Trigger Mode* on page 2-107).

Setting a trace start and end point

To set a trace start or end point (see *Trace start point and trace end point* on page 2-30), select the required option from the Set/Toggle Tracepoint List Selection dialog box:

Trace Start Point


Sets a trace start point on the selected address in the File Editor pane or memory pane. When you select **Trace Start Point**, an arrow  is placed in the left margin, next to the line of code you have selected.

For hardware targets, see *Setting trace ranges in conjunction with trace start and end points* on page 2-33 for a description of the tracing behavior when setting trace ranges in conjunction with trace start and end points.

————— Note —————

If you do not set a **Trace End Point** in combination with a **Trace Start Point**, all-inclusive trace information is returned from the start point onward.

Trace End Point

Signifies an instruction to turn tracing off at a specific address, ensuring that any ongoing trace collection stops. When you select **Trace End Point**, an arrow  is placed in the left margin next to the line of code you have selected.

For hardware targets, see *Setting trace ranges in conjunction with trace start and end points* on page 2-33 for a description of the tracing behavior when setting trace ranges in conjunction with trace start and end points.

————— Note —————

Setting a trace end point but no trace start point results in a different behavior. For an ETM-based system, no trace information is returned, and a warning is displayed in the **Cmd** tab of the Output pane.

Setting a trace range

When you set a trace range using the options described in this section, the end address is automatically set to 0xFFFFFFFF. To set a specific end address, you must use the corresponding end range point option.


You can set trace ranges in conjunction with trace start and end points. See *Setting trace ranges in conjunction with trace start and end points* on page 2-33 for a description of the tracing behavior in this situation.

Note

If you want to set an exclude trace range within a standard trace range, you must first set a specific end address for the standard trace range.


To set a trace range (see *Trace range* on page 2-31), select the required option in the Set/Toggle Tracepoint List Selection dialog box:


Start of Trace Range (Instruction Only)

Sets the start point for a range of addresses for which trace of program instructions only are captured. When you select this option, an arrow  is placed in the left margin next to the line of code you have selected.

After you have selected a start range point, the corresponding end-range point option, in this case **End of Trace Range (Instruction Only)**, becomes available in the List Selection dialog box. No other range-setting options are available until after you select an associated end-range point.

Start of Trace Range (Instruction and Data)

Sets the start point for a range of addresses for which trace of program instructions and data accesses are captured. When you select this option, an arrow  is placed in the left margin, next to the line of code you have selected.


After you have selected a start range point, the corresponding end-range point option, in this case **End of Trace Range (Instruction and Data)**, becomes available in the List Selection dialog box. No other range-setting options are available until after you select an associated end-range point. When you select an end-range point, an arrow  is placed in the left margin next to the line of code you have selected.


Start of Excluded Trace Range (Instruction and Data)

Sets the start point for a range of addresses for which trace of program instructions and data accesses are not captured. This option is the inverse of the option **Start of Trace Range (Instruction and Data)**, where the excluded range you set ensures that program instructions and data accesses are captured for all areas of your application except those within the range you specify.

Note

If the excluded range you specify contains a branch to another area of your application, that branched area is included in the trace capture if it has itself been marked for capture, or if no other points are set.

When you select this option, an arrow  is placed in the left margin next to the line of code you have selected.


After you have selected a start-range point, the corresponding end-range point option, in this case **End of Excluded Trace Range (Instruction and Data)**, becomes available in the List Selection dialog box. No other range-setting options are available until after you select an associated end-range point. When you select an end-range point, an arrow  is placed in the left margin next to the line of code you have selected.


Start of Excluded Trace Range (Data Only)

Sets the start point for a range of addresses for which trace of data accesses only are not captured. That is, program instructions for the range you specify are captured, and program instructions and data accesses for all areas outside that range are also captured.

————— Note —————

If the excluded range you specify contains a branch to another area of your application, the program instructions and data accesses of that branched area are also included in the trace capture if they have themselves been marked for capture, or if no other points are set.

When you select this option, an arrow  is placed in the left margin, next to the line of code you have selected.

After you have selected a start range point, the corresponding end-range point option, in this case **End of Excluded Trace Range (Data Only)**, becomes available in the List Selection dialog box. No other range-setting options are available until after you select an associated end-range point. When you select an end-range point, an arrow  is placed in the left margin next to the line of code you have selected.

Setting an ExternalOut Point

The ExternalOut points do not cause trace to be captured, but enable external signals to be generated by the compute logic of the ETM. The result of external outputs depends on the ASIC designer.

To set the ExternalOut points (see *ExternalOut Points* on page 2-32), select the required option from the Set/Toggle Tracepoint List Selection dialog box:

- **Set ExternalOut1 Point**
- **Set ExternalOut2 Point**
- **Set ExternalOut3 Point**
- **Set ExternalOut4 Point.**

Equivalent CLI commands used for unconditional tracepoints

When you set unconditional tracepoints using the Set/Toggle Tracepoint List Selection dialog box, RealView Debugger generates a TRACEINSTREXEC command with appropriate qualifiers.

The TRACEINSTREXEC command qualifiers that RealView Debugger uses when you set unconditional tracepoints are:

Trigger hw_out="Tracepoint Type=Trigger"

Trace Start Point

hw_out="Tracepoint Type=Start Tracing"

Trace Start Point (Instruction Only)

hw_out:"Tracepoint Type=Trace Start Point (Instr)"

Trace Start Point (Instruction and Data)

hw_out:"Tracepoint Type=Trace Start Point (Instr+Data)"

Trace End Point

For ETM-based targets, this is:

hw_out="Tracepoint Type=Stop Tracing"

Start of Trace Range (Instruction Only)

hw_out="Tracepoint Type=Trace Instr"

Start of Trace Range (Instruction and Data)

hw_out="Tracepoint Type=Trace Instr and Data"

Start of Excluded Trace Range (Instruction and Data)

hw_out="Tracepoint Type=Trace Instr",hw_not=addr

Start of Excluded Trace Range (Data Only)

hw_out="Tracepoint Type=Trace Instr and Data",hw_not=addr

ExternalOut Points

hw_out="Tracepoint Type=ExternalOut1"

hw_out="Tracepoint Type=ExternalOut2"

hw_out="Tracepoint Type=ExternalOut3"

hw_out="Tracepoint Type=ExternalOut4"

For more details on the TRACEINSTREXEC command, see the *RealView Developer Kit v2.2 Command Line Reference Guide*.

2.6.3 Setting a trace range for selected source lines or disassembly

You can set a trace range for selected lines of source or disassembly. To do this:

1. In the File Editor pane, highlight a range of source lines in the selected source code tab, or disassembly in the **Dsm** tab.
2. Right-click in the gray bar to the left of the source or disassembly to display the context menu.
3. Select **Set Trace Range** from the context menu.

RealView Debugger ensures the trace information is captured only for the range you have selected. Only program instructions in the range are captured. Any instructions executed outside the range are not captured. These discontinuities are represented as Trace Pause status lines in the Analysis window (see *Status lines in the Trace tab* on page 2-86).

CLI command used for setting an instruction-only trace range

The equivalent CLI command used is:

- in the **Dsm** tab:
TRACE,range *start_address..end_address*
- in a source code tab:
TRACE,range \MODULE\#*start_line*..\MODULE\#*end_line*

For more details on the TRACE command, see the *RealView Developer Kit v2.2 Command Line Reference Guide*.

2.6.4 Setting unconditional tracepoints with the Set/Edit Tracepoint dialog box

You can use the Set/Edit Tracepoint dialog box to set unconditional tracepoints manually. To do this, you only have to set the following parameters:

- tracepoint type
- tracepoint comparison type
- **when** (address or address range).

The following section, *ETM-based targets* on page 2-47, describes the values to use for each parameter of the Set/Edit Tracepoint dialog box, if you want to set the tracepoint equivalents described in *Using the Set/Toggle Tracepoint List Selection dialog box* on page 2-39.

For full details on using the Set/Edit Tracepoint dialog box, see *Using the Set/Edit Tracepoint dialog box* on page 2-55.

ETM-based targets

For an ETM-based target, Table 2-3 shows the values to use in the Set/Edit Tracepoint dialog box to set the tracepoint equivalents described in *Using the Set/Toggle Tracepoint List Selection dialog box* on page 2-39.

Table 2-3 Parameters for manually setting unconditional tracepoints on ETM-based targets

Type of tracepoint	Tracepoint Type setting	Tracepoint Comparison Type setting	when setting
Trigger	Trigger	Instr Exec	<i>address</i> or <i>address_range</i>
Trace Start Point	Start Tracing	Instr Exec	<i>address</i>
Trace End Point	Stop Tracing	Instr Exec	<i>address</i>
Trace Range ^a	Trace Instr	Instr Exec	<i>address_range</i>
Start of Trace Range (Instr Only) ^a	Trace Instr	Instr Exec	<i>address_range</i>
Start of Trace Range (Instr and Data) ^a	Trace Instr and Data	Instr Exec	<i>address_range</i>
Start of Excluded Trace Range (Instr and Data) ^a	Trace Instr	Instr Exec	\$NOT\$ <i>address_range</i>
Start of Excluded Trace Range (Data Only) ^a	Trace Instr and Data	Instr Exec	\$NOT\$ <i>address_range</i>
Trace ExternalOut1, Trace ExternalOut2, Trace ExternalOut3, or Trace ExternalOut4	ExternalOut1, ExternalOut2, ExternalOut3, or ExternalOut4	Instr Exec	<i>address</i> or <i>address_range</i>

- a. To set the equivalent end range address for these types, specify the address range as *start_address*..*end_address*. The end range address must be greater than the start range address. Use an end range address of 0xFFFFFFFF if you want to trace to the end of the image.

2.7 Setting conditional tracepoints

Conditional tracepoints enable you to test for various conditions that must be met before trace capture can begin. Conditional tracepoints are available only for ETM-based targets.

For an overview of the types of tracepoints you can set in RealView Debugger, see *Tracepoints in RealView Debugger* on page 2-29.

For more details on configuring the ETM for ETM-based targets, see *Configuring the ETM* on page 2-16.

2.7.1 Methods for setting conditional tracepoints

You can set conditional tracepoints using one of the following dialog boxes:

- Set/Edit Tracepoint dialog box (see *Using the Set/Edit Tracepoint dialog box* on page 2-55).
- Trace on X after Y [and/or Z] dialog box (see *Using the Trace on X after Y [and/or Z] dialog box* on page 2-60)
- Trace on X after Y executed N times dialog box (see *Using the Trace on X after Y executed N times dialog box* on page 2-61)
- Trace on X after A==B dialog box (see *Using the Trace on X after A==B dialog box* on page 2-63)
- Trace if A==B in X dialog box (see *Using the Trace if A==B in X dialog box* on page 2-65).

The Set/Edit Tracepoint dialog box creates a single tracepoint instance. The remaining dialog boxes create multiple tracepoint instances, which are chained together (see *Chaining tracepoints* on page 2-66 for more information).

The dialog boxes described in this section have a set of common parameters, described in *Common parameters for setting conditional tracepoints* on page 2-49. For parameters that are specific to a dialog box, see the section describing that dialog box.

When you set a tracepoint using these dialog boxes, RealView Debugger also generates the equivalent trace commands with qualifiers. The parameters that determine which trace command and qualifier is generated is also shown in the following sections. For more details, see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*.

2.7.2 Capturing instructions and data

If you want to capture any instructions and data, set a trace range with the Set/Edit Tracepoint dialog box to set a tracepoint of type **Trace Instr and Data**, and specify an address range. See *Tracepoint types* for details.

2.7.3 Common parameters for setting conditional tracepoints


In each of the dialog boxes that are available for setting tracepoints (see *Setting conditional tracepoints* on page 2-48), you can:

- set the tracepoint type from a drop-down list (see *Tracepoint types*)
- set the tracepoint comparison type from a drop-down list (see *Tracepoint comparison types* on page 2-50)
- enter the required addresses, address ranges, or data address comparison (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
- enter a value to test for (see *Entering data value comparisons* on page 2-53)
- select an external condition to test for (see *External Conditions* on page 2-54).

Tracepoint types

The tracepoint type specifies the output trigger test. This adds the `hw_out:"Tracepoint Type=type"` qualifier to the CLI command that is generated by RealView Debugger when it sets a tracepoint (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

The following tracepoint types are available:

Trigger Sets an explicit trigger point on the selected address. A trigger point enables you to capture trace before, after, or about the trigger point (see the **Collect trace...** options in *Trigger Mode* on page 2-107).
When you set a trigger point, an arrow  is placed in the left margin, next to the line of code you have selected.

Start Tracing

Starts tracing at the selected address.

———— Note ————

This type is available only on the Set/Edit Tracepoint dialog box, and only for ETM-enabled processors.

Stop Tracing

Stops tracing at the selected address.

———— **Note** ————

This type is available only on the Set/Edit Tracepoint dialog box.

Trace Instr Traces instructions only.

Trace Instr and Data

Traces both instructions and data.

ExternalOut1-4

Controls single-bit output signals from the ETM. Up to four signals are available. The ASIC manufacturer determines the availability and usage of these output signals. See your ASIC documentation for details.

Tracepoint comparison types

The tracepoint comparison type specifies the action that triggers the tracepoint. This determines the CLI command that is generated by RealView Debugger when it sets a tracepoint (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

The following tracepoint comparison types are available:

Instr Exec The address of each instruction that is presented to the execution unit is compared against the address you specify (even though the instruction might not be executed if its condition code evaluates to False).

This generates a TRACEINSTREXEC command.

Instr Fetch The address of each instruction fetched is compared against the address you specify.

This generates a TRACEINSTRFETCH command.

Data Read The meaning of this type depends on whether or not you specify a data value:

- If you do not specify data value, then the address of the data read from is compared against the address you specify.
- If you specify a data value, then the data value read from the specified address is compared against the value you specified.

This generates a TRACEDATAREAD command.

Data Write The meaning of this type depends on whether or not you specify a data value:

- If you do not specify data value, then address of the data written to is compared against the address you specify.
- If you specify a data value, then the data written to the specified address is compared against the value you specified.

This generates a TRACEDATAWRITE command.

Data Access The meaning of this type depends on whether or not you specify a data value:

- If you do not specify data value, then address of the data accessed, in either a read or write direction, is compared against the address you specify.
- If you specify a data value, then the data value accessed at the specified address, in either a read or write direction, is compared against the value you specified.

This generates a TRACEDATAACCESS command.

External Condition

The tracepoint is activated on one of the following events:

- external inputs 1 to 4
- EmbeddedICE watchpoints 1 and 2
- access to ASIC memory maps 1 to 16.

Up to four signals are available. The ASIC manufacturer determines the availability and usage of these input signals. See your ASIC documentation for details. Also, see *External Conditions* on page 2-54.


This generates a TRACEEXTCOND command.

Note


Not all tracepoint comparison types are available for all tracepoint units.

Entering addresses, address ranges, and data address comparisons

You can enter addresses, address ranges, and data address comparisons in the following ways:

- Click on the drop-down arrow  to:
 - choose from a Function List, Variable List, Module/File List, or Register List dialog box (see the chapter that describes working with browsers in the *RealView Developer Kit v2.2 Debugger User Guide* for instructions on how to use these dialog boxes)
 - select from your personal Favorites List
 - select from a list of previously-used expressions.

The options shown here depend on your debug target and connection.

- Type the required address, address range, or data address comparison into the text box. You can use the right arrow  to help with the syntax of the entry if required. The following options are available:

Address Range If you select this option when the text box is empty, RealView Debugger inserts `start..end` into the text box. Replace `start` with the start address and `end` with the end address.

 If you select this option when the text box contains a value, RealView Debugger takes this value as the start address, and inserts `..` after the value. Enter the end address.

Address Range by Length

If you select this option when the text box is empty, RealView Debugger inserts `start..+len` into the text box. Replace `start` with the start address and `len` with the required offset value in bytes.

If you select this option when the text box contains a value, RealView Debugger takes this value as the start address, and inserts `..+` after the value. Enter the required offset value in bytes.

NOT Address Compare

When you select this option, RealView Debugger inserts `NOT` into the text box. If the text box already contains an address or address range, `NOT` is inserted before it.

Otherwise, enter the address or address range after `NOT`, for example:

`NOT0x8000..0x8100`

You can use this option to set up excluded trace ranges (see *Setting a trace range* on page 2-42 and Table 2-3 on page 2-47).

This adds the `hw_not:addr` command qualifier (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

Autocomplete Range

This option generates an auto-range from an expression that you enter, which can be any of:

- A function name, where the generated address range is from the start-to-end of the function.
- A structure, where the generated address range is from the start-to-end of the structure.
- An array symbol, where the generated address range is from the start of the variable to the end, where the end is the *start+sizeof(var)*. For example, if the start address is `0x8000`, and the array size is 16 bytes, the end address is considered to be `0x8010` (that is, `0x8000+16`).


RealView Debugger filters the information down to only rows represented by the generated auto-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 of the function to the end. Similarly, enter a global variable to see the end-of-range address autocompleted as the variable storage address plus variable size.

Entering data value comparisons


You can enter values to compare against. This adds the `hw_dvalue:` command qualifier (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

Enter a value in the following ways:

- Click on the drop-down arrow  to:
 - choose from a Function List, Variable List, Module/File List, or Register List dialog box (see the chapter that describes working with browsers in the *RealView Developer Kit v2.2 Debugger User Guide* for instructions on how to use these dialog boxes)

- select from your personal Favorites List (see the chapter that describes working with browsers and favorites in the *RealView Developer Kit v2.2 Debugger User Guide*)
- select from a list of previously-used expressions.

The options shown here depend on your debug target and connection.

- Type the required value into the text box. You can use the right arrow  to help with the syntax of the entry if required. The following options are available:

Value Mask

The value mask enables you to specify individual bits to test when comparing values. Testing is performed on the following basis:

- a binary zero in the filter indicates that the bit is not tested
- a binary one in the filter indicates that the corresponding bit of the transfer is compared with the corresponding bit of the Data value.

When you select this option, RealView Debugger inserts `$MASK$=0xFFFFFFFF` into the text box. Enter the value you want to compare against, and edit the mask to the required value.

This adds the `hw_dmask:mask` command qualifier (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).


NOT Value Compare

When you select this option, RealView Debugger inserts `NOT` into the text box. If the text box already contains a value, `NOT` is inserted before it. Otherwise, enter the value after `NOT`, for example:

`NOT0x1000`

This adds the `hw_not:data` command qualifier (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

External Conditions

External conditions are application-specific, and correspond to the signals on your ASIC. See your ASIC documentation for details. The conditions are available only for the tracepoint comparison type **External Condition**. You must select the condition from the drop-down menu that is displayed when you click the drop-down arrow  of the **when** parameter.

These options add the `hw_in:"External Condition=condition"` command qualifier to the CLI command generated by RealView Debugger (see the description of the TRACEEXTCOND command in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

The following external conditions are available:

ExternalIn1-4

Up to four external inputs are available, depending on your ETM configuration. The ASIC manufacturer determines the availability and use of these output signals. External inputs can be any combination of logic that evaluates to True or False.

Watchpoint1-2

These resources are the watchpoint units in the EmbeddedICE macrocell. They are used when you set hardware watchpoints or hardware breakpoints. This functionality is only available when the signals from the EmbeddedICE are connected to the ETM. The ASIC manufacturer determines the availability and use of these output signals.

ASIC MemMap 1-16

These resources are only available if you have ASIC memory map decode hardware installed in your ASIC. Each implementation of ASIC memory map decode hardware has been defined by the ASIC manufacturer to return True when instructions are being fetched from a particular address range.

2.7.4 Using the Set/Edit Tracepoint dialog box

The Set/Edit Tracepoint dialog box, shown in Figure 2-10, enables you to set the parameters for single conditional tracepoint. The dialog shown is for an ETM-based target.

———— Note ————

You can also use this dialog box to edit and copy an existing tracepoint. See *Managing tracepoints* on page 2-68 for more details.

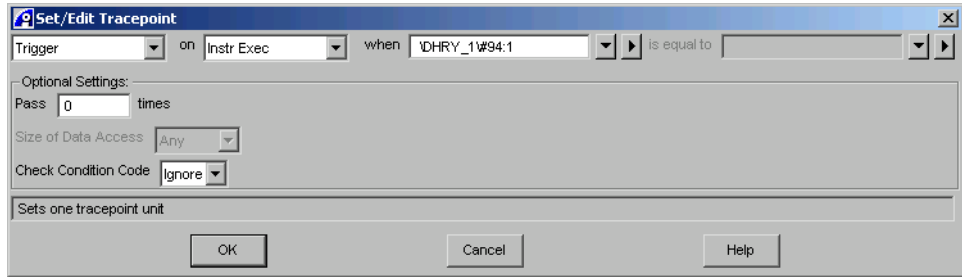


Figure 2-10 Set/Edit Tracepoint dialog box

Setting a single conditional tracepoint

To set a conditional tracepoint using the Set/Edit Tracepoint dialog box:

1. Display the dialog box using one of the following methods:
 - In the File Editor pane, right-click on the gray bar to the left of the source and select the **Set Detailed Tracepoint...** option from the context menu.
 - Select **Debug** → **Tracepoints** → **Set Tracepoint...** from the Code window main menu.
 - Select **Set Tracepoint...** from the Break/Tracepoint pane **Pane** menu.
 - Right-click anywhere in the Break/Tracepoint pane and select **Set Tracepoint...** from the context menu.
 - Right-click on a tracepoint in the Break/Tracepoint pane and select **Copy Break/Tracepoint...** from the context menu.
2. Select the required tracepoint type from the first drop-down list (see *Tracepoint types* on page 2-49).

3. Select the tracepoint comparison type from the second drop-down list (see *Tracepoint comparison types* on page 2-50). The tracepoint comparison type determines which of the remaining parameters you can set, shown in Table 2-4.

Table 2-4 Parameters available for each comparison type

Comparison type	Available parameters
Instr Exec	when (location specifier) Pass times Check Condition Code
Instr Fetch	when (location specifier) Pass times
Data Read	when (address or address range) is equal to (value) ^a Pass times Size of Data Access
Data Write	when (address or address range) is equal to (value) ^a Pass times Size of Data Access
Data Access	when (address or address range) is equal to (value) ^a Pass times Size of Data Access
External Condition	when (External Condition) Pass times

a. You do not have to enter a value for this parameter.

4. Specify the location or external condition:
 - If you selected any comparison type except **External Condition**, specify:
 - a line number in the source code, with or without a module name prefix
 - an address or address range (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
 - a function name, for example, Func_1

- a function entry point, for example, Func_1\@entry (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details on using the @entry symbol).
 - If you selected a comparison type of **External Condition**, specify the external condition you want to test for (see *External Conditions* on page 2-54).
5. If you selected a comparison type of **Data Read**, **Data Write**, or **Data Access**, specify the value you want to test against (see *Entering data value comparisons* on page 2-53). The tracepoint is activated if the location specified in step 4 contains this value. If you leave this value blank, then no data comparison occurs. Only accesses at the address, and of the type specified, are tested.
 6. If required, enter the number of times a point can be passed over before the specified condition is enabled (see *Pass times*).
 7. If you selected a comparison type of **Data Read**, **Data Write**, or **Data Access**, select the required data access size from the drop-down list (see *Size of Data Access* on page 2-59).
 8. If you selected a comparison type of **Instr Exec**, select the required **Check Condition Code** from the drop-down list (see *Check Condition Code* on page 2-59).
 9. Click **OK** to set the tracepoint and close the **Set/Edit Tracepoint** dialog box.

Additional information is displayed in the information box at the bottom of the Set/Edit Tracepoint dialog box.

Pass times

Pass is the number of times a point can be passed over before the specified condition is enabled. Enter the required pass into the text box. If you do not set this item, it defaults to 0, and the condition is executed each time the point is hit.

———— Note —————

If you specify a range in the when location field, then the ETM counters are decremented on every cycle executed in that range.

This adds the hw_passcount:count command qualifier (see the description of the trace commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

Size of Data Access

The size of the data transfer is compared against the address you specify. This adds the `hw_out:"Size of Data Access=size"` command qualifier to the CLI command that is generated by RealView Debugger when it sets a tracepoint (see the description of the trace data commands in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

This parameter is only available if you select the **Data Read**, **Data Write**, or **Data Access** comparison types (see *Tracepoint comparison types* on page 2-50).

The following options are available:

- Any** If you do not want to match on size of data access. This is the default.
- Halfword** To check both byte addresses in the halfword. This must be used, for example, if you are interested in byte accesses to either byte of a halfword.
- Word** To check all four byte addresses in the same word. This must be used, for example, if you are interested in byte accesses to any byte of a word.

Check Condition Code

Matches on the execution status of the address you specify. The address of each instruction that reaches the Execute stage of the pipeline is compared against the address you specify (it might not actually be executed if its condition code evaluates to False).

This adds the `hw_out:"Check Condition Code=code"` command qualifier to the CLI command that is generated by RealView Debugger when it sets a tracepoint (see the description of the TRACEINSTREXEC command in the *RealView Developer Kit v2.2 Command Line Reference Guide*).

This parameter is only available if you select the **Instr Exec** comparison type (see *Tracepoint comparison types* on page 2-50).

————— Note —————

This parameter is not available if you are using ETM hardware implementing ETM version 1.0 or 1.1.

The following options are available:

- Ignore** If you do not want to match on execution status. This is the default.
- Pass** To match only instructions that executed.

Fail To match only instructions that did not execute.

2.7.5 Using the Trace on X after Y [and/or Z] dialog box

The Trace on X after Y [and/or Z] dialog box, shown in Figure 2-11, enables you to define a tracepoint chain that is only active when one or more specified conditions have been met. For example, you can set a tracepoint chain that is only active when the first specified function has been executed but the second has not.

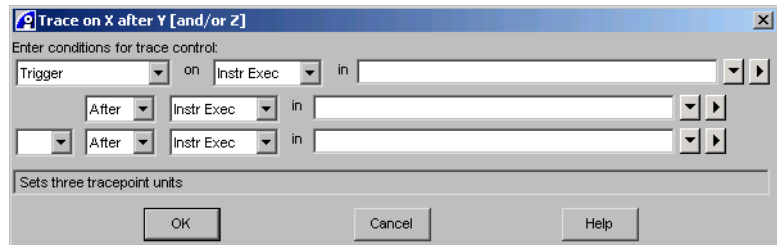


Figure 2-11 Trace on X after Y [and/or Z] dialog box

Setting a Trace on X after Y [and/or Z] tracepoint chain

To set a conditional Trace on X after Y [and/or Z] tracepoint:

1. Select **Debug** → **Tracepoints** → **Trace on X after Y [and/or Z]...** from the Code window main menu to display the Trace on X after Y [and/or Z] dialog box.
2. For the first tracepoint unit:
 - a. Select the type of tracepoint type that you want to set, for example **Trigger** (see *Tracepoint types* on page 2-49).
 - b. Select the tracepoint comparison type (see *Tracepoint comparison types* on page 2-50), for example **Data Access**.
 - c. Specify the location to test. This can be:
 - a line number in the source code, with or without a module name prefix
 - the address of an instruction, variable, or data value, or a range of addresses (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
 - a function name, for example, Func_1
 - a function entry point, for example, Func_1\@entry (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details on using the @entry symbol).

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

This tracepoint unit adds the `hw_and:"then-next"` command qualifier to the CLI command generated by RealView Debugger, and is the first tracepoint in the chain.

3. For the second tracepoint unit:
 - a. Select **After** or **Before** from the drop-down list:
 - select **After** if you want the tracepoint to trigger after the specified event has occurred
 - select **Before** if you want the tracepoint to trigger before the specified event has occurred.
 - b. Choose the tracepoint type and address range in the same way as for the first tracepoint unit.

This tracepoint unit adds the `hw_and:"then-prev"` command qualifier to the CLI command generated by RealView Debugger, and is the second tracepoint in the chain. If you selected **Before**, then it also adds the `hw_not:then` command qualifier.

4. If you want to specify a third tracepoint unit:
 - a. Select **AND** or **OR** from the first drop-down list.
 - b. Select **After** or **Before** from the second drop-down list.
 - c. Choose the tracepoint type and address range in the same way as for the first tracepoint unit.

This tracepoint unit adds the following command qualifiers to the CLI command generated by RealView Debugger, and is the third tracepoint in the chain:

- `hw_and:prev` command qualifier, if **AND** is selected
- `hw_and:"then-h1"` command qualifier, if **OR** is selected.
- `hw_not:then` command qualifier, if **Before** is selected.

5. Click **OK** to set the tracepoint as specified.

2.7.6 Using the Trace on X after Y executed N times dialog box

The Trace on X after Y executed N times dialog box, shown in Figure 2-12, enables you to set a tracepoint chain that becomes active when the secondary condition has been met a specified number of times. See *Setting up a conditional tracepoint* on page 2-149 for a detailed example.

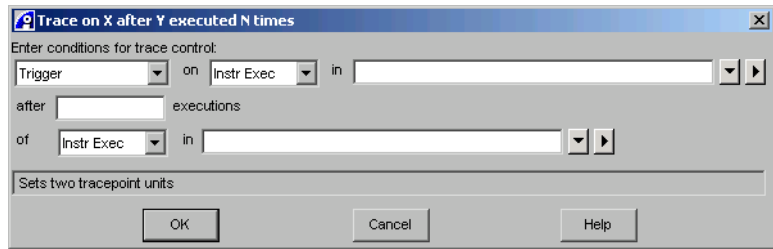


Figure 2-12 Trace on X after Y executed N times dialog box

Setting a Trace on X after Y executed N times tracepoint chain

To set a conditional Trace on X after Y executed N times tracepoint:

1. Select **Debug** → **Tracepoints** → **Trace on X after Y executed N times...** from the Code window main menu to display the Trace on X after Y executed N times dialog box.
2. For the first tracepoint unit:
 - a. Select the type of tracepoint type that you want to set, for example **Trigger** (see *Tracepoint types* on page 2-49).
 - b. Select the tracepoint comparison type (see *Tracepoint comparison types* on page 2-50), for example **Data Access**.
 - c. Specify the location to test. This can be:
 - a line number in the source code, with or without a module name prefix
 - the address of an instruction, variable, or data value, or a range of addresses (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
 - a function name, for example, Func_1
 - a function entry point, for example, Func_1\@entry (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details on using the @entry symbol).

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

This tracepoint unit adds the hw_and:"then-next" command qualifier to the CLI command generated by RealView Debugger, and is the first tracepoint in the chain.

3. Enter the required number of executions for the secondary condition.

This adds the `hw_pass:count` command qualifier to the CLI command generated by RealView Debugger for the second tracepoint in the chain.

4. For the second tracepoint unit:
 - a. Select the tracepoint comparison type (see *Tracepoint comparison types* on page 2-50), for example **Data Access**.
 - b. Specify the location to test. The location can be:
 - a line number in the source code, with or without a module name prefix
 - the address of an instruction, variable, or data value, or a range of addresses (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
 - a function name, for example, `Func_1`
 - a function entry point, for example, `Func_1\@entry` (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details on using the `@entry` symbol).

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

This tracepoint unit adds the `hw_and:"then-prev"` command qualifier to the CLI command generated by RealView Debugger, and is the second tracepoint in the chain.

5. Click **OK** to set the tracepoint as specified.

2.7.7 Using the Trace on X after A==B dialog box

This Trace on X after A==B dialog box, shown in Figure 2-13 on page 2-63, enables you to specify a tracepoint chain that only becomes active after a specified value is written to or read from a specified memory location. For example, you can set a tracepoint on a specified function being executed but only after zero has been written to a specified variable.

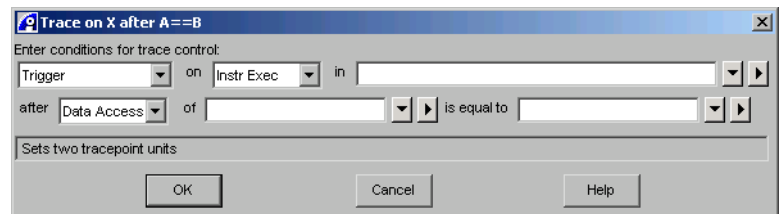


Figure 2-13 Trace on X after A==B dialog box

Setting a Trace on X after A==B tracepoint chain

To set a conditional Trace on X after A==B tracepoint:

1. Select **Debug** → **Tracepoints** → **Trace on X after A==B...** from the Code window main menu to display the Trace on X after A==B dialog box.
2. For the first tracepoint unit:
 - a. Select the type of tracepoint type that you want to set, for example **Trigger** (see *Tracepoint types* on page 2-49).
 - b. Select the tracepoint comparison type (see *Tracepoint comparison types* on page 2-50), for example **Data Access**.
 - c. Specify the location to test. The location can be:
 - a line number in the source code, with or without a module name prefix
 - the address of an instruction, variable, or data value, or a range of addresses (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
 - a function name, for example, Func_1
 - a function entry point, for example, Func_1\@entry (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details on using the @entry symbol).

This tracepoint unit adds the hw_and:"then-next" command qualifier to the CLI command generated by RealView Debugger, and is the first tracepoint in the chain.

3. Set the second tracepoint unit:
 - a. Select the tracepoint comparison type, for example **Data Access**.
 - b. Specify a variable name or data value address (see *Entering data value comparisons* on page 2-53).
 - c. Specify the value you want to compare the variable against (see *Entering data value comparisons* on page 2-53). You must enter a value in this field.

This tracepoint unit adds the hw_and:"then-prev" command qualifier to the CLI command generated by RealView Debugger, and is the second tracepoint in the chain.

4. Click **OK** to set the tracepoint as specified.

2.7.8 Using the Trace if A==B in X dialog box

This Trace if A==B in X dialog box, shown in Figure 2-14, enables you to set a tracepoint chain on a specified value being written to or read from a specified address at the same time as another condition is satisfied. For example, you can set a tracepoint on the value of a specified variable being altered by a specified function.

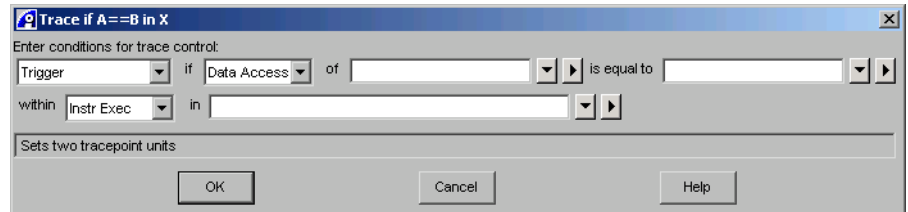


Figure 2-14 Trace if A==B in X dialog box

Setting a Trace if A==B in X tracepoint chain

To set a conditional Trace if A==B in X tracepoint:

1. Select **Debug** → **Tracepoints** → **Trace if A==B in X...** from the Code window main menu to display the Trace if A==B in X dialog box.
2. Set the first tracepoint unit:
 - a. Select the type of tracepoint that you want to set, for example **Trigger** (see *Tracepoint types* on page 2-49).
 - b. Select the tracepoint comparison type (see *Tracepoint comparison types* on page 2-50), for example **Data Access**.
 - c. Specify a variable name or data value address.
 - d. Specify the value you want to compare against (see *Entering data value comparisons* on page 2-53). You must enter a value in this field.

This tracepoint unit adds the `hw_and:"then-next"` command qualifier to the CLI command generated by RealView Debugger, and is the first tracepoint in the chain.

3. Set the second tracepoint unit:
 - a. Select the tracepoint comparison type, for example **Data Access**.
 - b. Specify the location to test. The location can be:
 - a line number in the source code, with or without a module name prefix

- the address of an instruction, variable, or data value, or a range of addresses (see *Entering addresses, address ranges, and data address comparisons* on page 2-52)
- a function name, for example, Func_1
- a function entry point, for example, Func_1\@entry (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details on using the @entry symbol).

This tracepoint unit adds the hw_and:"then-prev" command qualifier to the CLI command generated by RealView Debugger, and is the second tracepoint in the chain.

4. Click **OK** to set the tracepoint as specified.

2.7.9 Chaining tracepoints

You can create complex conditional tracepoints by chaining individual tracepoints together. You can create chained tracepoints in the following ways:

- Using the following dialog boxes:
 - Trace on X after Y [and/or Z] dialog box (see *Using the Trace on X after Y [and/or Z] dialog box* on page 2-60)
 - Trace on X after Y executed N times dialog box (see *Using the Trace on X after Y executed N times dialog box* on page 2-61)
 - Trace on X after A==B dialog box (see *Using the Trace on X after A==B dialog box* on page 2-63)
 - Trace if A==B in X dialog box (see *Using the Trace if A==B in X dialog box* on page 2-65).
- Using trace command qualifiers (see *Chaining tracepoints with command qualifiers* on page 2-66).

Chaining tracepoints with command qualifiers

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

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

Note

It is not recommended that you use the CLI to create chained tracepoints. If you want to use chained tracepoints in scripts, then create them using the tracepoint dialog boxes described in this section, and copy the commands generated by RealView Debugger.

2.8 Managing tracepoints

Whenever you set a tracepoint of any type, it is displayed in the Break/Tracepoints pane, shown in Figure 2-15.

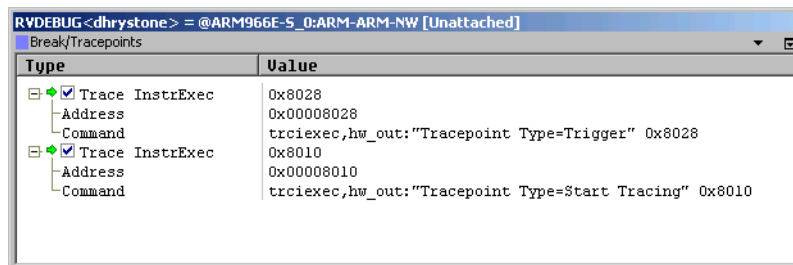



Figure 2-15 The Break/Tracepoints pane

Note

A conditional tracepoint might be displayed as two or more chained tracepoints in the Break/Tracepoints pane (see *Chaining tracepoints* on page 2-66).

To see the address and command details for a specific tracepoint, expand the display by clicking .

This section includes:

- *Editing tracepoints*
- *Copying tracepoints* on page 2-69
- *Disabling tracepoints* on page 2-69
- *Clearing tracepoints* on page 2-70
- *Using tracepoint favorites* on page 2-70
- *Finding source code corresponding to a tracepoint* on page 2-71.

2.8.1 Editing tracepoints

You can edit an existing tracepoint to change one or more parameters, or to attach conditions to an unconditional tracepoint. To edit a tracepoint:

1. In the Break/Tracepoints pane, right-click on the required tracepoint.
2. Select **Edit Break/Tracepoint...** from the context menu. This displays the Set/Edit Tracepoint dialog box (see *Using the Set/Edit Tracepoint dialog box* on page 2-55 for information on using this dialog box).

3. Edit the tracepoint as required, then click **OK** to confirm any changes and close the Set/Edit Tracepoint dialog box.

When you edit a tracepoint, the tracepoint command includes the modify command qualifier (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details).

———— **Note** ————

You cannot change the tracepoint type for tracepoints that succeed other tracepoints in a chain. For these tracepoints, the tracepoint type drop-down list is disabled, and a message is displayed in the Information box at the bottom of the Set/Edit Tracepoint dialog box informing you that this tracepoint is changed. All other fields can be edited as for independent tracepoints.




2.8.2 Copying tracepoints

You can use an existing tracepoint as the basis for creating a new tracepoint. To copy an existing tracepoint:

1. In the Break/Tracepoints pane, right-click on the tracepoint you want to use as a template for the new tracepoint.
2. Select **Copy Break/Tracepoint...** from the context menu. This displays the Set/Edit Tracepoint dialog box (see *Using the Set/Edit Tracepoint dialog box* on page 2-55 for information on using this dialog box).
3. Modify the tracepoint as required, then click **OK** to confirm any changes and close the Set/Edit Tracepoint dialog box.

2.8.3 Disabling tracepoints

To disable a tracepoint, unselect the checkbox ☐. When you disable a tracepoint, the arrow icons in the left margin that indicate tracepoints become fainter, to indicate that the tracepoint is disabled:

-  indicates that a trigger has been disabled
-  indicates that a trace start point or start of trace range has been disabled
-  indicates that a trace end point or end of trace range has been disabled.

You can re-enable the tracepoint by selecting the checkbox ☒.

When you disable a tracepoint that is chained with others to form a conditional tracepoint, those tracepoints that succeed it in the chain are also disabled. When you enable a Tracepoint that is linked with others to form a conditional tracepoint, the tracepoints that precede it in the chain are also enabled. In both cases, a warning is displayed in the **Cmd** tab of the Output pane.

2.8.4 Clearing tracepoints

To remove a tracepoint, right-click on the entry in the Break/Tracepoints pane and select **Clear** from the context menu.

If you clear a tracepoint that is chained with others to form a conditional tracepoint, those tracepoints that succeed it in the chain are also cleared, and a warning is displayed in the **Cmd** tab of the Output pane.

2.8.5 Using tracepoint favorites

You can set a tracepoint from a favorite you have saved previously, or you can add an existing tracepoint to the tracepoint Favorites List. The Favorites List is managed using the Favorites Chooser/Editor dialog box. For details on how to use the Favorites Chooser/Editor dialog box, see the chapter that describes working with browsers and favorites in the *RealView Developer Kit v2.2 Debugger User Guide*.

Setting a tracepoint from the Favorites List

To set a tracepoint from the Favorites List:

1. Display the Favorites Chooser/Editor dialog box as follows:
 - Select the following option from the Code window main menu:
Debug → Breakpoints → Set Break/Tracepoint from List → Break/Tracepoint Favorites...
 - Right-click in the Register pane, and select **Break/Tracepoint Favorites...** from the context menu.
2. Select the required tracepoint from the Favorites List.
3. Click **Set** to set the tracepoint, and close the Favorites Chooser/Editor dialog box.

Adding a tracepoint to the Favorites List


To add a tracepoint to the Favorites List:

1. Right-click on the tracepoint in the Register pane, and select **Break/Tracepoint Favorites...** from the context menu. The Favorites Chooser/Editor dialog box is displayed, and the selected tracepoint is inserted into the Add to List field.
2. Click **Add to List**. The tracepoint is added to the Break/Tracepoints Favorites List.
3. Click **Close** to close the Favorites Chooser/Editor dialog box.

2.8.6 Finding source code corresponding to a tracepoint

To locate the line of source code corresponding to a tracepoint:

1. Select **View → Break/Tracepoints** from the Code window main menu to display the Register pane.
2. Right-click on the required tracepoint entry in the Break/Tracepoints pane, and select **Show Code** from the context menu.

If this case, an arrow  is placed next to the line of source code in the File Editor pane to which the tracepoint corresponds.

2.9 Capturing trace output

This section describes how to capture trace output. It includes:

- *Configuring trace capture requirements*
- *Starting trace capture.*

2.9.1 Configuring trace capture requirements

If you have not already done so, configure your trace capture requirements as described in:

- *Configuring trace capture* on page 2-35
- *Setting unconditional tracepoints* on page 2-39
- *Setting conditional tracepoints* on page 2-48
- *Managing tracepoints* on page 2-68.

2.9.2 Starting trace capture

After you have configured your trace capture requirements, you must instruct RealView Debugger to begin tracing. To do this:

1. Select **View** → **Analysis Window** from the Code window main menu to display the Analysis window.
2. For ETM-based targets, ensure that **Tracing Enabled** is selected in the **Edit** menu of the Analysis window (see *Configuring trace options* on page 2-103).

———— **Note** ————

Tracing is enabled by default, but trace information is only returned if you have set a tracepoint or if you have selected an **Automatic Tracing Mode** from the **Edit** menu in the Analysis window.

3. Select **Debug** → **Run** to execute your application image.

For detailed instructions on how to use the Analysis window, see *Using the Analysis window* on page 2-73.

For examples of how to perform tracing with RealView Debugger, see *Examples of using trace in RealView Debugger* on page 2-134.

2.10 Using the Analysis window

This section describes the ways you can use the Analysis window. It contains the following sections:

- *Displaying the Analysis window*
- *Components of the Analysis window* on page 2-74
- *Viewing trace information* on page 2-79
- *Viewing source* on page 2-90
- *Viewing profiling information* on page 2-91
- *Configuring trace options* on page 2-103
- *Configuring view options* on page 2-110
- *Finding information* on page 2-112
- *Filtering captured information* on page 2-118
- *Saving and loading trace information* on page 2-127.

Also, see *Mapping Analysis window options to CLI commands and qualifiers* on page 2-130.

2.10.1 Displaying the Analysis window

To display the Analysis window, shown in Figure 2-16, select **View → Analysis Window** from the Code window menu. The **Trace** tab view is displayed by default.

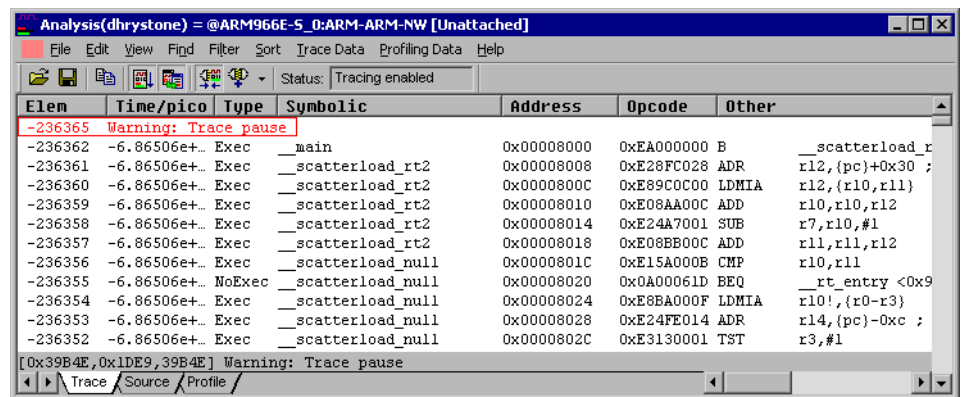


Figure 2-16 Example of the Analysis window

2.10.2 Components of the Analysis window

The components of the Analysis window, shown in Figure 2-16 on page 2-73, are described in the following sections:

- *Tabs*
- *Columns*
- *Main menus*
- *Context menu* on page 2-75
- *Toolbar* on page 2-77
- *Status bar* on page 2-78
- *Details view* on page 2-78.

Tabs

The Analysis window contains the following tabs:

Trace	Displays the trace data (see <i>Viewing trace information</i> on page 2-79).
Source	Displays the source code preceded by a single line giving the file name, line number(s) and function that each block relates to (see <i>Viewing source</i> on page 2-90).
Profile	Displays a statistical analysis of your trace information (see <i>Viewing profiling information</i> on page 2-91).

Columns

The Analysis window displays tracing and profiling information as a table of values. You can set which columns are displayed in both tabs using the options in the **Profiling Data** and **Trace Data** menus.

For information on the columns that are available in the **Trace** tab, and how to specify which tabs are displayed, see *Viewing trace information* on page 2-79. For information on the columns that are available in the **Profile** tab, and how to specify which tabs are displayed, see *Viewing profiling information* on page 2-91.

Main menus

The Analysis window main menu contains the following options:

File	Enables you to store and retrieve captured trace and profiling information. See <i>Saving and loading trace information</i> on page 2-127.
-------------	--

Edit	Enables you to set trace configuration options that are common to all trace captures you perform. See <i>Configuring trace options</i> on page 2-103.
View	Enables you to configure the way in which you want to view trace information. See <i>Configuring view options</i> on page 2-110.
Find	Enables you to locate a specific position within the captured trace output. See <i>Finding information</i> on page 2-112.
Filter	Enables you to filter the results of a trace capture that has already been performed. See <i>Filtering captured information</i> on page 2-118.
Sort	Enables you to sort the information in the Profile tab by a specified column. See <i>Sorting profiling information</i> on page 2-102.
Trace Data	Enables you to specify the columns you want to display or hide in the Trace tab. See <i>Viewing trace information</i> on page 2-79.
Profiling Data	Enables you to specify the columns you want to display or hide in the Profile tab. See <i>Viewing profiling information</i> on page 2-91.
Help	Provides access to the RealView Debugger online help.

Context menu

You can right-click in any tab to display the Analysis window context menu. The options in this menu are:

Track in Code Window (double right-click)

Select this option to track addresses in the code window. RealView Debugger locates the source or disassembly line in the appropriate File Editor pane tab that corresponds to the currently selected line in the Analysis window. The results of tracking are dependent on the tab you are accessing in the File Editor pane:

Src tab When you select a row of output representing an instruction in the Analysis window, RealView Debugger inserts a marker next to the corresponding source line.

Note

If the instruction you are selecting is at an address that does not correspond to one of your source files, no tracking occurs.

Dsm tab When you select a row of output representing an instruction in the Analysis window, RealView Debugger inserts a marker next to the corresponding disassembly line.

You can also track addresses in this manner by double-clicking on the desired row in the Analysis window.

———— **Note** —————

No tracking occurs if you select a row that does not represent an instruction, or does not have a data address value.

To turn off tracking, deselect the option **Track in Code Window (double-click)**.

Time Measure from Selected...

Displays the number of cycles from the selected line to the current line.

To use this feature, you must:

1. Select (single-click) the row representing the starting point from which you want to measure.
2. Right-click on the row representing the finishing point for the measurement.
3. Select **Time Measure from Selected...** from the context menu. An Information dialog box is displayed, for example, Figure 2-17.

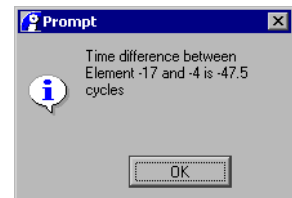


Figure 2-17 Time measurement dialog box

Time Measure from Trigger

Displays the time or number of cycles from the trigger to the selected line, depending on the Scale Time Units setting (see *Scale Time Units...* on page 2-111). For ETM-enabled hardware, the default is a time unit (Nanoseconds).

Find Next Searches the trace output for the next instance of the search item you have specified. See *Finding information* on page 2-112 for details on performing searches in the Analysis window.

Find Previous

Searches the trace output, from the current cursor position, for the previous instance of the search item you have specified. See *Finding information* on page 2-112 for details on performing searches in the Analysis window.

Copy Copies the selected text to the clipboard.

Toolbar

The Analysis window toolbar provides quick access to some of the more commonly-accessed menu items. It contains the following buttons:

**Load Trace Buffer from File...**

This button enables you to load a previously saved trace buffer from a file, which you can re-analyze. This option is useful in cases where you are performing a trace capture that takes a long time to reach the point of interest, and you do not want to have to repeat the process. You can also analyze the profiling information of the saved trace buffer even after you continue to make modifications to the source code.

**Save Trace Buffer to File...**

This button enables you to save the current trace information to a file. See *Save Trace Buffer to File...* on page 2-127 for information on using this dialog box.

**Copy**

Copies the selected text to the clipboard.

**Enable Trace**

This button globally enables tracing, and is selected by default. See *Tracing Enabled* on page 2-105 for more information.

**Track in Code Window**

When this button is clicked, automatic address tracking occurs. RealView Debugger locates the source or disassembly line in the appropriate File Editor pane tab that corresponds to the currently selected line in the Analysis window.

**Connect to Analyzer**

This button enables you to connect to and disconnect from the analyzer.



Cycle Connections

This button is not used.

Status bar

The status bar displays the current state of the trace. This can be one of:

- Tracing enabled** The **Tracing Enabled** option in the **Edit** menu is selected. This message only occurs with trace targets that support the **Tracing Enabled** option.
- Tracing disabled** The **Tracing Enabled** option in the **Edit** menu is not selected. This message only occurs with trace targets that support the **Tracing Enabled** option.
- Not connected** There is no trace support for the target, or the target is not connected.
- Ready** Trace functionality is now available. This message only occurs with targets that do not support the **Tracing Enabled** option in the **Edit** menu.
- Updating** The Analysis window is being updated with data from the trace target.

Details view

The details view is located directly above the tabs at the bottom of the Analysis window, as shown in Figure 2-18. To display status-bar information, select the option **Show Details View** from the **View** menu.

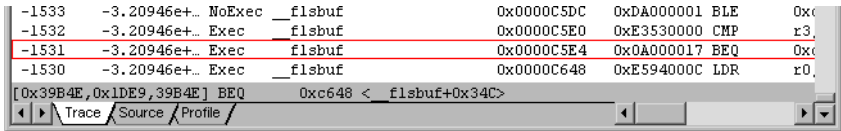


Figure 2-18 Details view

The details view provides different information depending on the tab currently displayed:

[Data in buffer,Data read,Data RealView Debugger reads] Disassembly

Data in buffer

The amount of data in the trace buffer.

Data read The amount of data currently read.

Data RealView Debugger reads

The amount of data read by RealView Debugger.

Disassembly This field shows information corresponding to the currently selected line:

- For a line of trace output, a disassembly of the instruction or data value. This is the same information that is shown in the **Other** column.
- For an error or warning message, then that message appears here. If the line contains the message no source available, then this field shows the disassembly of the instruction immediately following that line.
- For an interleaved source line, then this field shows the disassembly of the instruction immediately following the current group of interleaved source lines.
- For a line showing inferred registers or an instruction boundary, then this field shows the disassembly of the associated instruction.
- For a line showing a function boundary, then this field shows the disassembly of the first instruction for the associated function.

See *Rows in the Trace tab* on page 2-83 for details of how to display interleaved source, inferred registers, and instruction and function boundaries.

2.10.3 Viewing trace information

To view the trace data, click on the **Trace** tab. This tab enables you to view:

- Captured trace data, and shows:
 - the symbolic representation of instructions that have been traced
 - the symbolic representation of data that has been traced, interleaved between the traced instructions, if you have specified data capture in the trace configuration
 - the disassembly of traced instructions and data, with branches to functions showing the symbolic name of the function.

The display of this information is controlled using the options in the Columns section of the **Trace Data** menu (see *Columns in the Trace tab* on page 2-80).

- Additional information that helps you to interpret the captured trace:
 - interleaved source
 - inferred registers

- function boundaries
- instruction boundaries.

The display of this information is controlled using the options in the Rows section of the **Trace Data** menu (see *Rows in the Trace tab* on page 2-83).

———— **Note** —————

You can also hide and show the traced instructions and data as required.

The **Trace** tab also shows error and warning messages where appropriate (see *Status lines in the Trace tab* on page 2-86).

Columns in the Trace tab

The columns displayed in the **Trace** tab depend on the items you have selected in the Columns section of the **Trace Data** menu (see *Displaying columns in the Trace tab* on page 2-83). The following columns are available:

Elem	<p>Displays the position of each element within the trace buffer, where the value can represent either:</p> <ul style="list-style-type: none"> • An index within the trace buffer. • A cycle number, if your trace capture device supports cycle-accurate tracing. To display cycle numbers in this column you must select Cycle accurate tracing in the Configure ETM dialog box (see <i>Cycle accurate tracing</i> on page 2-24). <p>This column is displayed by default.</p>
Time/unit	<p>Displays the timestamp value. To change the format of the values, select View → Scale Time Units from the menu.</p> <p>This column shows information only when either Enable Timestamps or Cycle accurate tracing is selected in the Configure ETM dialog box (see <i>Enable Timestamping</i> on page 2-24 and <i>Cycle accurate tracing</i> on page 2-24). To scale between times and cycle numbers, select View → Define Processor Speed for Scaling... from the menu.</p> <p>This column is displayed by default.</p>
+Time	<p>Displays a delta timestamp value, indicating the time taken between the previous instruction and the current instruction. To change the format of the values, select View → Scale Time Units from the menu.</p>

This column shows information only when either **Enable Timestamps** or **Cycle accurate tracing** is selected in the Configure ETM dialog box (see *Enable Timestamping* on page 2-24 and *Cycle accurate tracing* on page 2-24).

This column is not displayed by default.

Type

Displays the access type of the current element, which can be any of:

Bus	Bus state change.
Code	Code access (fetch).
Data	Data access (read or write).
DMA	<i>Direct Memory Access</i> (DMA) transfer operation.
Exec	Instruction was executed.
Folded	Folded branch (folded by branch prediction unit and never issued).
FoldNoEx	Folded branch (folded by branch prediction unit and never issued) was not executed.
Int	Interrupt vectoring.
Instr	Instructions on targets where the trace does not contain executed flags.
<p style="text-align: center;">———— Note —————</p> <p style="text-align: center;">The instruction might or might not have been executed.</p>	
NoExec	An instruction has not been executed.
Pin	Pin state change.
PreF	Prefetch (so not executed).
Prob	External probe state change.

———— Note —————

If an access type is prefixed by R, this indicates a read access. A W prefix indicates a write access.

This column is displayed by default.

Symbolic

Displays the symbolic position information for the current element, and takes one of the following forms:

- `source_symbol_name+offset`

For example, `Arr_2_Glob+0x65` might be a data access to the variable address `Arr_2_Glob`, with an offset of `0x65`.

- `source_symbol_name[\[~]#line_number[. .#line_number]]`
`source_symbol_name`
This can be any symbolic information, including a function, module, variable, or low-level symbol
~ This means that the corresponding instruction, and the instructions corresponding to the symbols with the same line number, implement the same line of code. For example, instructions that implement a branch or loop.
#line_number[. .#line_number]
This means that the corresponding instructions span the specified lines of source code.

For example, your trace output might have the following sequence of symbols:

`Func_2\#159`
`Func_2\~#159`
`Func_2\#161. .#164`

The instructions corresponding to the symbols `Func_2\#159` and `Func_2\~#159` both implement the source code at line 159 in the function `Func_2()`.

The instruction corresponding to the symbol `Func_2\#161. .#164` spans lines 161 to 164 of the source in the function `Func_2()`.

This column is displayed by default.

Address	Displays the address of the instruction or data accessed. This column is displayed by default.
Data/Hex	Displays data values in hexadecimal form. This column is not displayed by default.
Data/Dec	Displays data values in decimal form. This column is not displayed by default.
Opcode	Displays the opcode of the instruction accessed. This column is displayed by default.
Other	For instructions, the disassembled instruction is displayed. For data, the display has the following format: <code><Data> 'character' hexvalue ['character' hexvalue ...]</code> <code>character</code> is a printable character or a C-style escape code.

hexvalue is the hexadecimal value of the character if it is neither a printable character, nor an escape code.

For example:

<Data> '\f' 0x03 '\0' '\0'

This column is displayed by default.

Count Displays the number of times a particular address was accessed.
This column is not displayed by default.

Displaying columns in the Trace tab

The options that are selected in the Columns section of the **Trace Data** menu control what information is displayed in the **Trace** tab. Table 2-5 summarizes the relationship between items in the **Trace Data** menu and the columns that are displayed:

Table 2-5 Trace Data menu items and column names

Trace Data Menu item	Column name
Position	Elem
Absolute Time	Time/unit
Relative Time	+Time
Access Type	Type
Address as Value	Address
Address as Symbol/Line	Symbolic
Data Value in Hex	Data/Hex
Data Value in Decimal	Data/Dec
Opcode	Opcode
Interpretation of Data/Opcode	Other
Count of Hits	Count

Rows in the Trace tab

You can control the type of trace information displayed by selecting the required option(s) from the **Trace Data** menu:

All Trace Displays all entries in the trace buffer, regardless of type.

Instruction Boundaries

Displays instruction boundaries, divided by ruler lines. This is useful if you want to see loaded and stored data associated with the instruction that loaded or stored the data. This also shows the traced instructions, even if the **Instructions** option is not selected.

Instructions Displays all instructions.

Data Displays all data.

Function Boundaries

Displays additional rows giving the function boundaries.

————— **Note** —————

Function boundaries show where a function was called. Where the compiler inlines function calls, function boundaries are not shown.

If you display **Inferred Registers** and **Function Boundaries** but no other instruction records, then this view shows values passed into and returned from functions. Therefore, you can track function behavior. When you are running AAPCS-compliant code on an ARM processor registers r0-r3 contain function parameters, and r0 contains a function's return value.

Interleaved Source

Displays the source lines associated with each trace buffer entry interleaved with the buffer entry. The corresponding code is displayed after each buffer entry. Where several successive buffer entries are associated with the same or overlapping sets of source lines, the source line is shown after the first buffer entry and then suppressed. Source code that is executed repeatedly is shown for each execution.

Inferred Registers

Displays inferred register values interleaved with the trace data, as shown in Figure 2-19 on page 2-85. It also shows the instruction boundaries, even if the **Instruction boundaries** option is not selected.


Inferred register values are inferred from the traced instructions, and are not read from the registers. This means that not all register values are always known.

Note

The Inferred Registers view is only available if you are using an ARM architecture-based processor. In addition, traced instructions must be shown in the **Trace** tab. That is, when any of the **Instructions** and **Instruction boundaries** options is selected.

Analysis(dhrystone) = @ARM966E-5_0:ARM-ARM1-NW [Unattached]

File Edit View Find Filter Sort Trace Data Profiling Data Help

 Status: Tracing enabled

Elem	Time/pico	Type	Symbolic	Address	Opcode	Other									
-186		Exec	Proc_3\~#358	0x000080F0	0xE280200C	ADD r2,r0,#0xc									
R0	-----	R1	-----	R2	-----	R3	-----	R4	-----	R5	-----	R6	-----	R7	-----
R8	-----	R9	-----	R10	-----	R11	-----	R12	00000000	SP	-----	LR	00008148	PC	000080F4
NZCV	0	STATE	ARM												
-185		Exec	Proc_3\~#358	0x000080F4	0xE59F0228	LDR r0,0x8324 <DHRV_1\#146>									
R0	-----	R1	-----	R2	-----	R3	-----	R4	-----	R5	-----	R6	-----	R7	-----
R8	-----	R9	-----	R10	-----	R11	-----	R12	00000000	SP	-----	LR	00008148	PC	000080F8
NZCV	0	STATE	ARM												
-184		Exec	Proc_3\~#358	0x000080F8	0xE5901000	LDR r1,[r0,#0]									
R0	-----	R1	-----	R2	-----	R3	-----	R4	-----	R5	-----	R6	-----	R7	-----
R8	-----	R9	-----	R10	-----	R11	-----	R12	00000000	SP	-----	LR	00008148	PC	000080FC
NZCV	0	STATE	ARM												
-183		Exec	Proc_3\~#358	0x000080FC	0xE3A0000A	MOV r0,#0xa									
R0	0000000A	R1	-----	R2	-----	R3	-----	R4	-----	R5	-----	R6	-----	R7	-----
R8	-----	R9	-----	R10	-----	R11	-----	R12	00000000	SP	-----	LR	00008148	PC	00008100
NZCV	0	STATE	ARM												
-182		Exec	Proc_3\~#358	0x00008100	0xEB0003A3	BL Proc_7 <0x8F94>									
R0	0000000A	R1	-----	R2	-----	R3	-----	R4	-----	R5	-----	R6	-----	R7	-----
R8	-----	R9	-----	R10	-----	R11	-----	R12	00000000	SP	-----	LR	00008104	PC	00008F94
NZCV	0	STATE	ARM												
-181		Exec	Proc_7	0x00008F94	0xE1A03000	MOV r3,r0									
R0	0000000A	R1	-----	R2	-----	R3	0000000A	R4	-----	R5	-----	R6	-----	R7	-----
R8	-----	R9	-----	R10	-----	R11	-----	R12	00000000	SP	-----	LR	00008104	PC	00008F98
NZCV	0	STATE	ARM												
[0x200000,0x1B91,200000] ADD r2,r0,#0xc															

Trace Source Profile

Figure 2-19 Example of inferred registers

The register data is shown following the instruction opcode, and shows the values in the registers after the instruction has executed. Register data is displayed in the following way:

- Register names are gray.
- Unchanged register values are black.
- Changed register values are blue.
- Unknown register values are represented by -----. If a register value became unknown in the current cycle, this appears in blue to indicate a changed register value.

The amount of information that can be inferred depends on the amount of information contained in the trace. If you have not traced data, few register values can be inferred. With full data trace, most register values can be inferred.

———— **Note** —————

When any of the **Instructions** and **Instruction Boundaries** options is selected, the inferred register values show the values contained in the registers after the instruction was executed.

When no instruction rows are displayed but **Function Boundaries** are displayed, the inferred register values shown are the values contained in the registers before the instruction at the function boundary was executed.

You can use the menu items at the bottom of the **Trace Data** menu to instruct RealView Debugger to display:

- only those registers where the value is known
- only those register where the value has changed between instructions
- a narrower view of the registers in the display.

Only Known Registers

Enables you to view only those registers whose values are known.

This option is only available when **Inferred Registers** is selected.

Only Changing Registers

Enables you to view only those registers where the values have changed between instructions.

This option is only available when **Inferred Registers** is selected.

Narrow Register View

Enables you to view six registers on a line, instead of eight.

This option is only available when **Inferred Registers** is selected.

Status lines in the Trace tab

Some rows of the returned trace output in the Analysis window are for status-only purposes, and provide information about the processor cycle. These status lines are displayed in red, and can be any of the following messages:

- *Error messages* on page 2-87
- *Warning messages* on page 2-89.

Error messages

The Error messages that might be displayed in the **Trace** tab are:

Error: Synchronization Lost

Indicates that RealView Debugger has detected trace data that does not correspond to the image loaded into the debugger, and therefore cannot be decoded.

Error: ETM FIFO Overflow

Indicates that tracing was temporarily suspended because the ETM FIFO buffer became full. When this occurs, there is a discontinuity of returned trace information.

Error: Coprocessor data transfer of unknown size

When tracing data, RealView Debugger executed an unrecognized coprocessor memory access instruction, and the decompressor could not deduce the amount of data transferred by the instruction. Decompression of data tracing, and data address tracing, stop until appropriate synchronization points are found in the trace data.

Error: Data synchronization lost following FIFO overflow

Some versions of the ARM ETM can cause corrupt data trace after a FIFO overflow has occurred. If the decompressor sees a case where this is likely to have happened, it outputs this message, and suppresses data and data address tracing until it can resynchronize.

Error: Unexpected exception

The instruction has marked an exception, but the exception address does not appear to be a valid exception address.

Error: Instruction not known

The decompressor was not in sync for this instruction, but later discovered that this instruction was an exception.

Error: Incorrect synchronization address

An address broadcast for synchronization did not match that being maintained by the decompressor.

Error: Instruction data overflowed end of buffer

The data for the instruction is not in the buffer. This can occur when trace capture has stopped because it filled the buffer between the instruction being traced and its data being traced. All available data addresses and data are traced.

Error: The next instruction was traced as a branch

The instruction on the next line is not a branch, but the ETM traced it as a branch. This usually indicates that the wrong image is loaded into the debugger, or the code is self-modifying.

Error: The next instruction was not traced as an indirect branch

The instruction on the next line is an indirect branch, but the ETM did not trace it as an indirect branch. This usually indicates that the wrong image is loaded into the debugger, or the code is self-modifying.

Error: The next instruction was traced as a memory access instruction

The trace from the ETM indicated that the instruction on the next line read some data from memory, or wrote some data to memory, but the instruction is not a memory access instruction. This usually indicates that the wrong image is loaded into the debugger, or the code is self-modifying. Decompression of data tracing and data address tracing stops until appropriate synchronization points are found in the trace data.

Error: The next instruction should have been executed unconditionally

The trace from the ETM indicated that the instruction on the next line failed its condition code test, so was not executed, but the instruction is one that must be executed unconditionally. This usually indicates that the wrong image is loaded into the debugger, or the code is self-modifying.

Error: Corrupt address in trace data

The trace data contains an impossible address. This only occurs as a result of a hardware problem (such as a faulty connector).

Error: The next instruction was not traced as a branch

The current instruction is a branch but the trace does not indicate this. This usually indicates that the wrong image is loaded into the debugger, or the code is self-modifying. Decompression of data tracing and data address tracing stop until appropriate synchronization points are found in the trace data.

Error: The next instruction could not be read

The memory containing the traced instruction could not be read. This might occur if the application image attempts to execute a region of unreadable memory, in which case the instruction aborts with a Prefetch Abort. It might also occur if trace is decoded while the image is still running, and the image attempts to execute code outside the loaded image.

Warning messages

The Warning messages that might be displayed in the **Trace** tab are:

Warning: Debug State

Indicates that tracing was suspended for several processor cycles because the processor entered debug state.

Warning: Trace Pause

Indicates that tracing was temporarily suspended because of the trace conditions that have been set. Trace Pause represents the period of execution between the areas you have defined to be traced.

Warning: Instruction address synchronization has been restored

This message occurs after a problem in which instruction address synchronization has been lost. It indicates that the decompressor has found a point at which it can resume decompressing instruction addresses.

Warning: Unable to trace Jazelle state, trace data ignored

The ETM detected the processor entering Jazelle® (bytecode) state. The decompressor is unable to decompress Jazelle bytecode execution, so all trace output is suppressed until the processor leaves Jazelle state.

Warning: Data address synchronization restored

This message occurs after a problem in which data address synchronization has been lost. It indicates that the decompressor has found a point at which it can resume decompressing data addresses.

Warning: Data synchronization restored

This message occurs after a problem in which data synchronization has been lost. It indicates that the decompressor has found a point at which it can resume decompressing data.

Warning: Memory address unknown, insufficient trace data

This warning only occurs near the beginning of the decoded trace when the trace buffer (not the FIFO) of the trace capture hardware has overflowed. It means that there has not yet been a complete memory access address in the trace data, and therefore the trace decoder cannot calculate the address of a data access. The ETM outputs a complete address on the first data access traced, and repeats this every 1024 cycles after this, if there are data accesses to be traced. To reduce buffer usage, other memory addresses are output relative to the last full memory

address. If the buffer overflows, and the complete address is lost, the decoder cannot calculate data addresses that occur before the next full data address is emitted.

Warning: Data suppression protected ETM FIFO from overflow

This warning occurs when you have enabled data suppression (see *FIFO overflow protection* on page 2-22), and the ETM has to suppress the output of data in an attempt to prevent an ETM FIFO overflow. This is usually followed by the Warning: Data synchronization restored status line when data trace is restored.

2.10.4 Viewing source

The **Source** tab in the Analysis window displays the source lines associated with each trace buffer entry. The buffer entries themselves are not displayed.

2.10.5 Viewing profiling information

Click the **Profile** tab in the Analysis window to view a statistical analysis of your trace information. You can use this tab to analyze control-flow information (branches), measure the time it takes to execute certain functions, and view call-graph data.

Note

The profile information is less accurate and in some cases might be incorrect if the captured trace is not continuous. To generate meaningful profiling information use trace start and end points at the boundaries of the functions you want to profile instead of, for example, setting ranges. It is important that you take into account which parts of your application image have been traced when interpreting profile information. For example, semihosting can affect the profiling results.

An example of the **Profile** tab is shown in Figure 2-20.

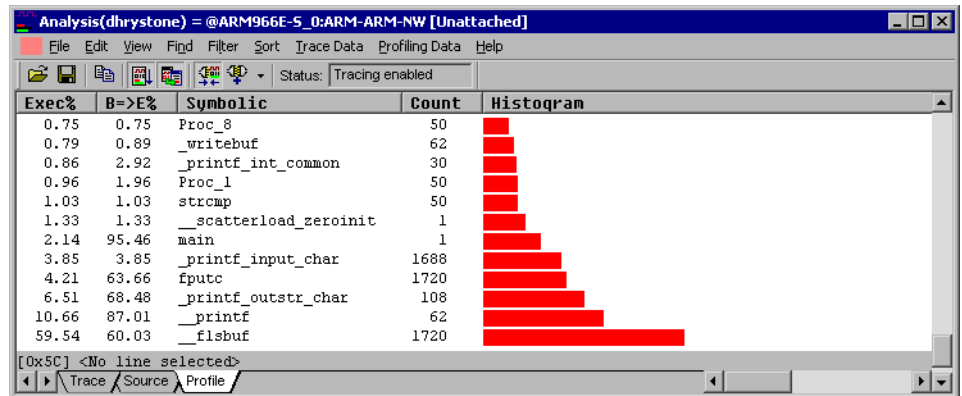


Figure 2-20 Profile tab

The information displayed in this tab depends on the options you have selected in the **Profiling Data** menu. The following sections describe the available views:

- *Columns in the Profile tab* on page 2-92
- *Call-graph data in the Profile tab* on page 2-95
- *Interpretation of profiling data* on page 2-101
- *Collection of profiling data* on page 2-102
- *Sorting profiling information* on page 2-102.

Columns in the Profile tab

The columns displayed in the **Profile** tab depend on the items you have selected in the Columns section of the **Profiling Data** menu (see *Displaying columns in the Profile tab* on page 2-94).

The following columns are available in the **Profile** tab:

1st	Displays the element number representing the first access to each function. This column is not displayed by default.
Exec%	Displays, as a percentage of the whole, the time spent in the range of this function or module, where the entire trace buffer represents 100%. This represents the PC in the range of the function itself, and does not include time spent in the descendents of the function. This column is displayed by default.
<hr/> <p style="text-align: center;">Note</p> <hr/> <p>Inlined functions count towards the time of the function into which they are inlined.</p> <hr/>	
B=>E%	Displays, as a percentage of the whole, the time elapsed from the beginning to the end of the range. This includes time spent in the descendents of the function. This column is displayed by default.
Type	Displays the type of range. This is usually Func , indicating a function. Where there are no functions, for example if the code is in assembly language, the range type is Module . This column is not displayed by default.
Symbolic	Displays the name of the function or module that is profiled. This column is displayed by default
Address	Displays the address of the symbol that is profiled. This column is not displayed by default
Exec/cycle	Displays the total time spent in execution(s) of this function. This column is not displayed by default.
B=>E	Displays the total beginning to end time of all executions of this function. This column is not displayed by default.

B=>E Avg Displays the average of all the individual times spent on the execution of this function.

This column is not displayed by default.

Min B =>E Displays the minimum time spent executing the function. This is especially useful when a particular function is executed several times, but for different tasks, and you want to see the lowest value of the execution times involved. The time is displayed in the format that is currently selected for analysis (see *Scale Time Units...* on page 2-111).

This column is not displayed by default

Max B =>E Displays the maximum time spent executing the function. This is especially useful when a particular function is executed several times, but for different tasks, and you want to see the highest value of the execution times involved. The time is displayed in the format that is currently selected for analysis (see *Scale Time Units...* on page 2-111).

This column is not displayed by default.

Count Shows the number of times that the function was entered and exited.

This column is displayed by default.

Note

If the trace begins or ends within a function, then that instance of the function is not counted. For an instance of a function to be counted, both entry to and exit from the function must be traced.

Histogram Displays the **Histogram**. You can view the histogram as a linear or a logarithmic function. By default, the histogram is displayed as a linear function. You can view the histogram as a logarithmic function of the Exec% or B=>E value by selecting **Use Logarithmic Scale for Histogram** in the **Profiling Data** menu.

This column is displayed by default.

Figure 2-21 on page 2-94 shows an example histogram.

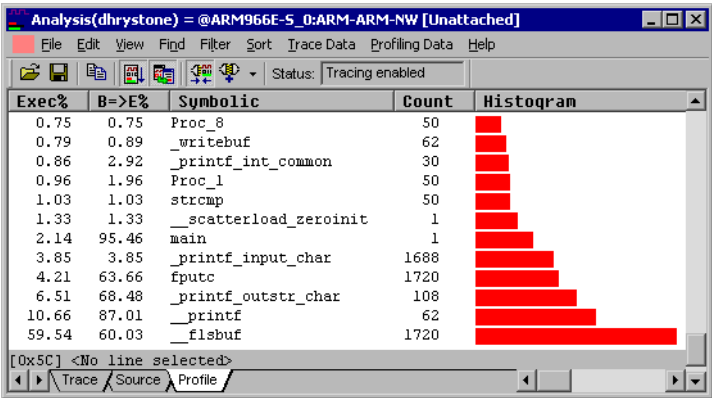


Figure 2-21 Example histogram

Note

The histogram is not available if timing information is not present. If you want to view profiling information, you must enable either timestamping or cycle-accurate tracing, or both. If you have an ETMv3 core, then instruction count-based profiling information is shown, unless you enable one of these options. You can configure both of these options using the Configure ETM dialog box (see *Configuring the ETM* on page 2-16). Better results in profiling are generally obtained with timestamping enabled.

See *Enable Timestamping* on page 2-24 and *Cycle accurate tracing* on page 2-24 for detailed information on these options.

Displaying columns in the Profile tab

The options that are selected in the Columns section of the **Profiling Data** menu control what information is displayed in the **Profile** tab. Table 2-6 summarizes the relationship between items in the **Profiling Data** menu and the columns that are displayed:

Table 2-6 Profiling Data menu items and column names

Profiling Data menu item	Column name(s)
First Instance	1st
Time% In Self	Exec%
Time% Including Children (B=>E)	B=>E%
Range Type	Type

Table 2-6 Profiling Data menu items and column names (continued)

Profiling Data menu item	Column name(s)
Range Symbol	Symbolic
Address as Value	Address
Exec/B=>E/B=>E Average	Exec/cycl
	B=>E
	B=>E Avg
Count of Calls	Count
Min/Max Times	Min B=>E
	Max B=>E
Histogram View	Histogram

Call-graph data in the Profile tab

Call-graph data enables you to view a list of parents and/or children for each function. The data for the function being profiled is colored black, and the histogram for that function, if present, is colored red.

When parents and/or children are displayed, a ruler line separates each group of function, parents and children.

You can view Call-graph data by selecting one or both of the following options from the **Profiling Data** menu:

Parents of Function

Displays the parents of the function. A parent is a function that makes a call to the function being profiled. The **Exec%** and **B=>E%** values shown for parents are the times spent in the child when called from this parent.

Parents are displayed before the function line, and the Symbolic information is indented. The data for parents is a light gray color, and the parent histograms, if present, are colored blue.

This option is disabled by default.

Children of Function

Displays the children of the function. A child is a function that is called by the function being profiled. The **Exec%** and **B=>E%** values shown for children are time spent in this function when called from the parent.

Children are displayed after the function line, and the Symbolic information is indented. The data for children is a light gray color, and the child histograms, if present, are colored green.

This option is disabled by default.

Note

Where the compiler inlines function calls, these inline functions are not shown as children.

Figure 2-22 shows example call-graph data.

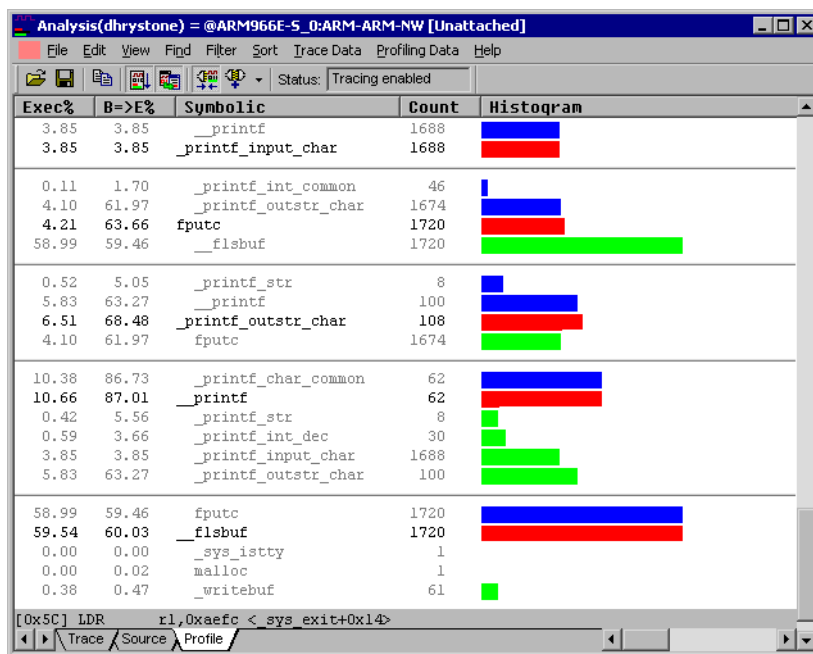


Figure 2-22 Example of call-graph information

The precise meaning of the call-graph values depends on the Data Interpretation option you have selected. See *Interpretation of profiling data* on page 2-101 for information on how to select these options. Available options are:

- *Parent/Child %ages Relative to Whole Time*
- *Parent/Child %ages Relative to Function B=>E* on page 2-98
- *Parent/Child %ages Relative to Parent/Child B=>E* on page 2-100.

Note

B=>E refers to *Beginning to End*.

Parent/Child %ages Relative to Whole Time

This is the default view. When this option is selected, the values shown for **Exec%** and **B=>E%** for parents and children are shown as a percentage of the whole time traced. Figure 2-23 gives an example of call-graph data for a function with its parents and children.

Exec%	B=>E%	Symbolic	Count
2.36	3.15	Proc_1	39
2.43	3.24	Proc_2	110
4.79	6.39	Proc_6	149
0.53	0.53	Func_2	29
1.07	1.07	Func_3	149

Figure 2-23 Example call-graph data (Parent/Child %ages Relative to Whole Time)

In this example, when **Parent/Child %ages Relative to Whole Time** is selected:

- The **Exec%** column shows that:
 - 2.36% of the total execution time was spent in code of the function Proc_6 when called from the parent Proc_1.
 - 2.43% of the total execution time was spent in code of the function Proc_6 when called from the parent Proc_2.
 - 4.79% of the total execution time was spent in code of the function Proc_6.
 - 0.53% of the total execution time was spent in code of the function Func_2 when called as a child from Proc_6.
 - 1.07% of the total execution time was spent in code of the function Func_3 when called as a child from Proc_6.

- The **B=>E%** column shows that:
 - 3.15% of the total execution time was spent in calls to function Proc_6 and its children when called from the parent Proc_1.
 - 3.24% of the total execution time was spent in calls to function Proc_6 and its children when called from the parent Proc_2.
 - 6.39% of the total execution time was spent in calls to function Proc_6 and its children.
 - 0.53% of the total execution time was spent in code of the function Func_2 and its children when called as a child from Proc_6.
 - 1.07% of the total execution time was spent in code of the function Func_3 and its children when called as a child from Proc_6.
- The **Count** column shows that:
 - There were 39 calls from the function Proc_1 to the function Proc_6.
 - There were 110 calls from the function Proc_2 to the function Proc_6
 - There were 149 calls to the function Proc_6.
 - There were 29 calls to the function Func_2 from the function Proc_6.
 - There were 149 calls to the function Func_3 from the function Proc_6.

Parent/Child %ages Relative to Function B=>E

When this option is selected, the values shown for **Exec%** and **B=>E%** for parents and children are shown relative to the total B=>E time of the function. Figure 2-24 gives an example of call-graph data for a function with its parents and children.

Exec%	B=>E%	Symbolic	Count
36.93	49.30	Proc_1	39
38.03	50.70	Proc_2	110
4.79	6.39	Proc_6	149
8.29	8.29	Func_2	29
16.74	16.74	Func_3	149

Figure 2-24 Example call-graph data (Parent/Child %ages Relative to Function B=>E)

In this example, when **Parent/Child %ages Relative to Function B=>E** is selected:

- The **Exec%** column shows that:
 - 36.93% of the total time spent in Proc_6 and its children was spent in Proc_6 when called from Proc_1.

- 38.03% of the total time spent in Proc_6 and its children was spent in Proc_6 when called from Proc_2.
- 8.29% of the total time spent in Proc_6 and its children was spent in Func_2 when called from Proc_6.
- 16.74% of the total time spent in Proc_6 and its children was spent in Func_3 when called from Proc_6.

Note

These four figures add up to 100% because the total time spent in Proc_6 and its children is divided between the time spent in Proc_6 when called from its parents, and the time spent in the children when called from Proc_6. This is because, in this example, there are no grandchildren.

- The **B=>E%** column shows that:
 - 49.30% of the total time was spent in Proc_6 and its children when Proc_6 was called from Proc_1.
 - 50.70% of the total time was spent in Proc_6 and its children when Proc_6 was called from Proc_2.
 - 8.29% of the total time spent in Proc_6 and its children was spent in Func_2 and its children when called from Proc_6.

Note

This figure is the same as that in the **Exec%** column only because there are no grandchildren.

- 16.74% of the total time spent in Proc_6 and its children was spent in Func_3 and its children when called from Proc_6.

Note

This figure is the same as that in the **Exec%** column only because there are no grandchildren.

Note

The value for the parents always totals 100%. The total time is split between the parents.

- See *Parent/Child %ages Relative to Whole Time* on page 2-97 for details of the **Count** column.

Parent/Child %ages Relative to Parent/Child B=>E

When this option is selected, the values shown for **Exec%** and **B=>E%** for parents and children are shown relative to the B=>E time of the parents and children. Figure 2-25 gives an example of call-graph data for a function with its parents and children.

Exec%	B=>E%	Symbolic	Count
10.30	13.74	Proc_1	39
38.40	51.20	Proc_2	110
4.79	6.39	Proc_6	149
50.00	50.00	Func_2	29
16.74	16.74	Func_3	149

Figure 2-25 Example call-graph data (Parent/Child %ages Relative to Parent/Child B=>E)

In this example, when **Parent/Child %ages Relative to Parent/Child B=>E** is selected:

- The **Exec%** column shows that:
 - 10.30% of the total time spent in Proc_1 and its children was spent in Proc_6 when called from Proc_1.
 - 38.40% of the total time spent in Proc_2 and its children was spent in Proc_6 when called from Proc_2.
 - 50.00% of the total time spent in Func_2 and its children was spent in Func_2 when called from Proc_6
 - 16.74% of the total time spent in Func_3 and its children was spent in Func_3 when called from Proc_6.
- The **B=>E%** column shows that:
 - 13.74% of the total time spent in Proc_6 and its children was spent in Proc_6 and its children when called from Proc_1.
 - 51.20% of the total time spent in Proc_6 and its children was spent in Proc_6 and its children when called from Proc_2.
 - 50.00% of the total time spent in Proc_6 and its children was spent in Func_2 and its children when called from Proc_6.

———— **Note** ————

This figure is the same as that in the **Exec%** column only because there are no grandchildren.

- 16.74% of the total time spent in Proc_6 and its children was spent in Func_3 and its children when called from Proc_6.

Note

This figure is the same as that in the **Exec%** column only because there are no grandchildren.

- See *Parent/Child %ages Relative to Whole Time* on page 2-97 for details of the **Count** column.

Interpretation of profiling data

The options in the Data Interpretation section of the **Profiling Data** menu are:

Ignore Holes in Trace

Instructs RealView Debugger to ignore holes in the trace data caused by discontinuities. Discontinuities might occur, for example, when the processor is in debug state, or when tracing is disabled. When this option is selected, holes are not included in the whole time used to calculate the values for **Exec%** and **B=>E%**.

Use Logarithmic Scale for Histogram

When this option is selected, the length of the histogram bar is calculated using a logarithmic function of the **Exec%** or **B=>E%** value. When this option is disabled, a simple linear scale is used.

Parent/Child %ages Relative to Whole Time

Enables you to specify the value that time percentages are calculated relative to. When this option is selected, the values shown for **Exec%** and **B=>E%** for parents and children are shown as a percentage of the whole time traced. This setting is the default. See *Parent/Child %ages Relative to Whole Time* on page 2-97 for detailed information on how to interpret call-graph data when this option is selected.

Parent/Child %ages Relative to Function B=>E

Enables you to specify the value that time percentages are calculated relative to. When this option is selected, the values shown for **Exec%** and **B=>E%** for parents and children are shown relative to the total B=>E time of the function. See *Parent/Child %ages Relative to Function B=>E* on page 2-98 for detailed information on how to interpret call-graph data when this option is selected.

Parent/Child %ages Relative to Parent/Child B=>E

Enables you to specify the value that time percentages are calculated relative to. When this option is selected, the values shown for **Exec%** and **B=>E%** for parents and children are shown relative to the B=>E time of the parents and children. See *Parent/Child %ages Relative to Parent/Child B=>E* on page 2-100 for detailed information on how to interpret call-graph data when this option is selected.

Collection of profiling data

The options in the Collection section of the **Profiling Data** menu are:

Sum Profiling Data

Instructs RealView Debugger to sum multiple runs together, instead of displaying the data for individual runs separately. If you want to clear the accumulated profiling data and begin collecting data for a new series of runs, select the **Zero Profiling Data** option.

For an example of how to use this option, see *Capturing profiling information* on page 2-143.

Zero Profiling Data

This option is only available when you select the **Sum Profiling Data** option. It instructs RealView Debugger to clear the accumulated profiling data and display the profiling data from the current trace only.

Sorting profiling information

The options in the **Sort** menu enable you to sort the information in the **Profile** tab by a specified column. Information can be sorted in ascending or descending order. There are two ways you can change the sorting order of the Analysis window output:

- Select one of the **By...** options in the **Sort** menu that determine which column is to be used as the sort key. You can also select the **Reverse Sort** option to reverse the order of the selected sort.
- Click on the column header for the column you want to sort by. If you click on the same column header again, the sorting order is reversed.

———— Note ————

If you click on a column header to sort by a particular column, the corresponding item in the **Sort** menu is automatically selected. If you click on the column header again to reverse the order, the item **Reverse Sort** is automatically selected.

The complete **Sort** menu options are as follows:

By Time Sorts the output by the **Exec** column. This is the default sort option.

By Address Sorts the output by the **Address** column.

By Name Sorts the output by the **Symbolic** column.

By Access Type

Sorts the output by the **Type** column.

By Count Sorts the output by the **Count** column.

By First Instance

Sorts the output by the **1st** column.

By Total B=>E Time

Sorts the output by the **B=>E** column.

By Average B=>E Time

Sorts the output by the **Avg B=>E** column.

By Minimum B=>E Time

Sorts the output by the **Min B=>E** column.

By Maximum B=>E Time

Sorts the output by the **Max B=>E** column.

Reverse Sort

Reverses the order of the sort you have selected. To return to ascending-order sorting (the default) on the specified column, you must deselect this option.

See *Capturing profiling information* on page 2-143 for a detailed example of how you can use RealView Debugger to capture profiling information.

2.10.6 Configuring trace options

This section describes how you use the **Edit** menu of the Analysis window to set trace configuration options that are common to all trace captures you perform.

————— **Note** —————

The availability of some of these menu options, and any dialogs that might be displayed when you select them, are dependent on your trace capture hardware, your target processor, and the state of some configuration options.

The checkmarks next to some options in the **Edit** menu indicate the default settings, which vary depending on the hardware you are using.

The complete menu options are:

- *Copy*
- *Connect Analyzer*
- *Disconnect Analyzer* on page 2-105
- *Tracing Enabled* on page 2-105
- *Configure Analyzer Properties...* on page 2-105
- *Set Trace Buffer Size...* on page 2-105
- *Buffer Full Mode* on page 2-106
- *Trigger Mode* on page 2-107
- *Data Tracing Mode* on page 2-108
- *Automatic Tracing Mode* on page 2-108
- *Set/Edit Event Triggers* on page 2-109
- *Clear All Event Triggers* on page 2-109
- *Physical to Logical Address Mapping...* on page 2-109.

Note

Make sure that RealView Debugger has finished updating the trace buffer, before you set a trace configuration option.

Copy

Copies the selected text to the clipboard.

Connect Analyzer

Connects to an analyzer.

Note

For ETM-based targets, you can use the **Logic_Analyzer Vendor** flag in the Connection Properties window to connect to an analyzer instead of using this option. For RealView Trace, you have to configure the flag manually. See the chapter that describes using RealView Trace in the *RealView ICE User Guide* for information on how to do this.

Disconnect Analyzer

Disconnects from an Analyzer. This option is grayed out if you are not connected to an analyzer. To connect, use the **Connect Analyzer...** option in the Analysis window **Edit** menu.

Tracing Enabled

This option globally enables tracing, and is selected by default. Deselect this option to globally disable tracing. This is useful, for example, if you want to stop capturing trace information so that you can view the current contents of the buffer before the image stops executing.

If you stop tracing during image execution, all captured information up to that point is returned to the Analysis window. You can reselect this option to restart tracing at any time during image execution.

Configure Analyzer Properties...

This option enables you to configure the analyzer settings. For an ETM-enabled target, this option displays the Configure ETM dialog box (see *Configure ETM dialog box interface components* on page 2-19).

This option is grayed out if you are not connected to an analyzer.

Set Trace Buffer Size...

This option enables you to set the size of the trace buffer. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-26.

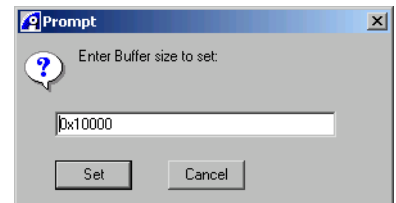


Figure 2-26 Setting the size of the trace buffer

Enter the required maximum buffer size, in decimal or hexadecimal, then click **Set**.

Note

Some trace capture hardware might not support a variable buffer size, and always stores the maximum number of cycles. Other hardware might support only certain set buffer sizes, and selects the buffer size closest to that requested. See the documentation that accompanies your analyzer for details.

Buffer Full Mode

Enables you to configure the behavior of RealView Debugger when the trace buffer is full. Available options are:

Stop Processor on Buffer Full

This option instructs RealView Debugger to stop the target processor executing when the trace buffer becomes full.

Stop Collecting on Buffer Full

This option instructs RealView Debugger to stop the trace information being collected when the trace buffer becomes full. The target processor is not stopped.

If you want the processor to stop whenever the trace buffer becomes full, you must select the option **Stop Processor on Buffer Full**.

Continue Collecting on Buffer Full

This option instructs RealView Debugger to continue collecting trace information while the processor continues to run, even after the trace buffer becomes full. In this case, older information is cleared from the buffer as new information enters it.

Note

This option is overridden if you have specified a trigger and the trigger is reached.

Note

For ETM targets, **Continue Collecting on Buffer Full** is set by default and cannot be changed.

Trigger Mode

Enables you to configure whether trace information is collected before, around or after the trigger (see *Trigger* on page 2-29). You can also instruct RealView Debugger to stop the target processor when the trigger is hit. Available options are:

Collect Trace Before Trigger

This option is the default. It instructs RealView Debugger to capture all trace information prior to the trigger position.

Note

The position of the trigger might be approximate, depending on the characteristics of your trace capture device. The trigger itself might or might not be present in the resulting trace.

Collect Trace Around Trigger

This option instructs RealView Debugger to capture all trace information around the trigger position.

If you do not specify trace start and end points, then half of the trace information prior to reaching the trigger, and half of the trace information after reaching the trigger, is captured.

If you have placed the trigger between trace start and end points, then the amount of trace information collected before and after the trigger depends on the position of the trigger relative to the trace start and end points.

Note

The position of the trigger might be approximate, depending on the characteristics of your trace capture device.

Collect Trace After Trigger

This option instructs RealView Debugger to capture all trace information after the trigger position.

Note

The position of the trigger might be approximate, depending on the characteristics of your trace capture device. The trigger itself might or might not be present in the resulting trace.

Stop Processor on Trigger

When this option is selected, RealView Debugger programs the ETM to stop the target processor when the trigger is hit.

Data Tracing Mode

Enables you to specify the type of information the ETM traces for data transfer instructions. Select one of the following values:

- | | |
|---------------------|---|
| Address Only | Use this option when you want the data transfer address, but no information about the value(s) transferred. This option is the default. |
| Data Only | Use this option when you want information about the value(s) transferred, but not the data transfer address. |

Data and Address

Use this option when you require both the data transfer address and information about the value(s) transferred.

Note

This option is available only for ETM-based targets. It does not enable data tracing, but only selects what type of information is stored when data trace is enabled. You can instruct RealView Debugger to capture addresses, data, or both in the following ways:

- Select the appropriate Automatic Tracing Mode (see *Automatic Tracing Mode*)
 - Set the required tracepoint type for individual tracepoints as you set them (see *Configuring trace capture* on page 2-35 for information on setting tracepoints). This overrides the Automatic Tracing Mode option you have selected.
-

Automatic Tracing Mode

Enables you to collect trace information without requiring you to configure any tracepoints. When this mode is set, you only have to execute your application image to generate trace information. When the processor stops, either by itself or by manual intervention, the trace information currently in the buffer is returned to the Analysis window.

Note

When a tracepoint is set, Automatic Tracing Mode is overridden and tracing is performed according to the tracepoint(s).

Use this option to specify which type of Automatic Tracing Mode you want to use, or to disable Automatic Tracing Mode. Available options are:

Off (Use tracepoints)

This option disables Automatic Tracing Mode. When this option is selected, you must use tracepoints to capture trace information (see *Tracepoints in RealView Debugger* on page 2-29).

Instructions Only

This option is the default. RealView Debugger automatically traces on instructions.

Data Only

This option is available only for ETMv3 targets, and is grayed out for all other targets. It instructs RealView Debugger to automatically trace on data only.

Instructions and Data

This option instructs RealView Debugger to automatically trace on both instructions and data.

Set/Edit Event Triggers

This option enables you to set or edit event triggers.

———— Note ————

This option is disabled for ETM-based targets.

Clear All Event Triggers

This option clears all event triggers set using the Set/Edit Event Triggers dialog box.

———— Note ————

This option is disabled for ETM-based targets.

Physical to Logical Address Mapping...

This option enables you to configure the address and signal controls of your target processor. It is enabled only if your target configuration supports this functionality.

Note

This option is disabled for ETM-based targets.

2.10.7 Configuring view options

This section describes how you use the **View** menu of the Analysis window to configure the way in which you want to view trace information.

The complete menu options are:

- *Update*
- *Clear Trace Buffer*
- *Code Window Tracking*
- *Show Details View*
- *Show Position Relative to Trigger*
- *Scale Time Units...* on page 2-111
- *Define Processor Speed for Scaling...* on page 2-111
- *Automatic Update on New Buffer* on page 2-112.

Update

This option refreshes the information in the Analysis window.

Clear Trace Buffer

This option clears the information in the trace buffer.

Code Window Tracking

Instructs the Code window to track to the location that is currently selected in the Analysis window.

Show Details View

Displays status-bar information in the details view at the bottom of the window (see *Status bar* on page 2-78). To hide the status-bar details, deselect this option.

Show Position Relative to Trigger

Displays the current position relative to the trigger point. RealView Debugger numbers elements from $-M$ to $+N$, where 0 is the trigger point. If you set a trigger point, this option is enabled automatically, and the relative position is shown by default.

If this option is disabled and there is no trigger point specified, the way in which elements are numbered depends on the hardware that you are using.

Note

This option is disabled for ETM-based targets. If you are using an ETM-based target, the relative position is always shown.

Scale Time Units...

Enables you to change the format in which time information is displayed in the Analysis window. When you select this option, a List Selection dialog box is displayed. Select one of the following time formats, then click **OK**:

- **Default**
- **Picoseconds**
- **Nanoseconds**
- **Microseconds**
- **Milliseconds**
- **Seconds**
- **Cycles.**

Note

The default format depends on both your trace capture hardware and your target processor. ETM-enabled processors have a default time format of nanoseconds.

If your default format is cycles, then you must define the processor speed for scaling if you select any other display format. If your default format is not cycles, then you must define the processor speed for scaling if you select cycles as the display format. See *Define Processor Speed for Scaling...* for information on how to do this.

Define Processor Speed for Scaling...

Enables you to specify the clock speed of the processor you are using, to set the scaling between cycles and timestamp values in the current buffer, as shown in the **Time/unit** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-27 on page 2-112.

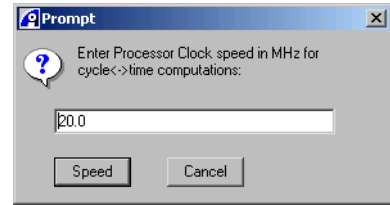


Figure 2-27 Defining processor speed

Enter a clock speed in MHz, then click **Speed**.

Automatic Update on New Buffer

This option instructs the Analysis window to be cleared of its contents, and updated automatically with new information when a new buffer load of trace data is returned.

2.10.8 Finding information

This section describes the **Find** menu options you can use to locate a specific position within the captured trace output. Each search is performed in a downwards direction starting at the cursor position. The menu options are described in the following sections:

- *Find Trigger* on page 2-113
- *Find Position Match...* on page 2-113
- *Find Time Match...* on page 2-114
- *Find Raw Address Match...* on page 2-115
- *Find Address Expression Match...* on page 2-116
- *Find Data Value Match...* on page 2-117
- *Find Symbol Name Match...* on page 2-117
- *Find Next Match* on page 2-118
- *Find Previous Match* on page 2-118.

Find Trigger

Locates the row of trace output representing the trigger point within your code. For details on setting triggers, see *Setting unconditional tracepoints* on page 2-39.

Note

If no trigger point is found, but you have set a trigger, then:

1. Increase the size of the trace buffer (see *Set Trace Buffer Size...* on page 2-105 for details).
2. Select **Target** → **Reload Image to Target** from the Code window main menu to reload the image without clearing your tracepoints.
3. Run the application again to recapture the trace information.

Find Position Match...

Enables you to locate a specific element number within the trace buffer (see the description of the **Elem** column in *Columns in the Trace tab* on page 2-80). When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-28.

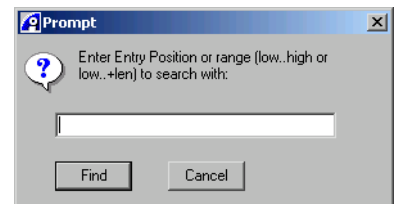


Figure 2-28 Finding a position match

Enter one of the following types of information, then click **Find**:

Entry position

Specify an element number, in decimal or hexadecimal format. Only an exact match is returned.

Entry position range

Specify a range of element numbers, where RealView Debugger displays the first found value within that range. The range you specify can be either:

- *low..high*, such as 1..10, where RealView Debugger locates the first occurrence of any Elem value within the range 1 to 10.

- *low..+len*, such as 40..+10, where RealView Debugger locates the first occurrence of any Elem value within the range 40 to 50. The *len* of 10 represents the offset value. You can also enter a negative value range, such as -10..10.

Find Time Match...

Enables you to locate a specific timestamp value within the **Time/unit** column (or, in the case of the **Profile** tab, the **Exec%** column). When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-29.

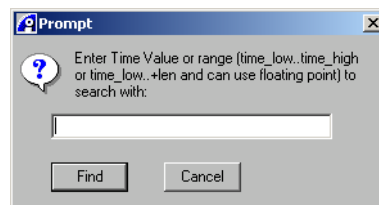


Figure 2-29 Finding a time match

Enter one of the following types of information, then click **Find**:

Time value Specify a timestamp value, in the format currently used for the **Time/unit** column. Only an exact match is returned. See *Scale Time Units...* on page 2-111 for details on changing the format in which time information is displayed in the Analysis window.

Time range Specify a range of timestamp values, in the format currently used for the **Time/unit** column. (See *Scale Time Units...* on page 2-111 for details on changing the format in which time information is displayed in the Analysis window.) In this case, RealView Debugger displays the first found value within that range. The range you specify can be either:

- *time_low..time_high*, such as -100..-50, where RealView Debugger locates the first occurrence of a timestamp value within the range -100 to -50.
- *time_low..+len*, such as -100..+10, where RealView Debugger locates the first occurrence of a timestamp value within the range -100 to -90. The *len* of 10 represents the offset value.

————— **Note** —————

- You can use floating point values, such as -50.5.

- Depending on your system solution, the **Time/unit** column might contain cycle numbers instead of timestamp values. In this case, you can search using cycle numbers.

Find Raw Address Match...

Enables you to locate a specific address within the **Address** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-30.

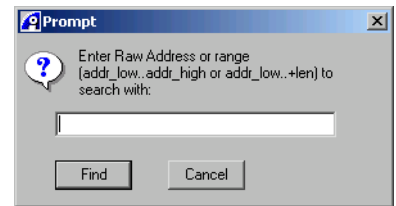


Figure 2-30 Finding a raw address match

Enter one of the following types of information, then click **Find**:

Raw address value

Specify an address value, in decimal or hexadecimal format. Only an exact match is returned.

Raw address range

Specify a range of address values, where RealView Debugger displays the first found address within that range. The range you specify can be either:

- addr_low..addr_high*, such as `0x00008E50..0x0000926C`, where RealView Debugger locates the first address that is found within that range.
- addr_low..+len*, such as `0x00008E50..+10`, where RealView Debugger locates the first address within the range `0x00008E50` to `0x00008E5A`. The *len* of 10 represents the offset value.

Note

To search for a specific address by entering a symbolic expression, you must use the option **Find Address Expression Match....**

Find Address Expression Match...

Enables you to locate a specific address, within the **Address** column, that corresponds to an expression you enter. An address expression can be a function, structure, or array symbol. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-31.

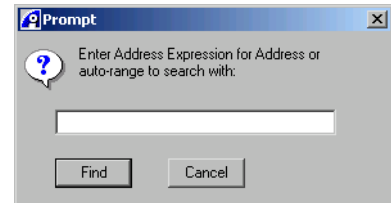


Figure 2-31 Finding an address expression match

Enter one of the following types of information, then click **Find**:

Address expression

Specify an address expression, which can be one of:

- a function name
- a structure name
- an array symbol.

Auto-range An auto-range of address values is generated from an expression that you enter, which can be one of:

- A function name, where the generated address range is from the start to the end of the function.
- A structure, where the generated address range is from the start-to-end of the structure.
- An array symbol, where the generated address range is from the start of the variable to the end, and the end address is the *start+sizeof(var)*. For example, if the start value of where the array is stored in memory is 0x8000, and the array size is 16 bytes, the end address is considered to be 0x8010 (that is, 0x8000+16).

RealView Debugger displays the first found address value within the auto-range that is generated.

Find Data Value Match...

Enables you to locate a specific data value that is read from, or written to, memory, within the **Data** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-32.

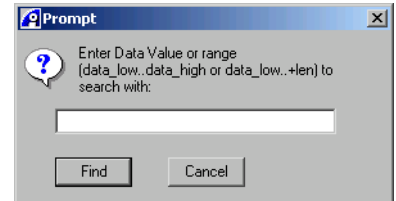


Figure 2-32 Finding a data value match

Enter one of the following types of information, then click **Find**:

Data value Specify a data value, in decimal or hexadecimal format.

Data range Specify a range of data values, where RealView Debugger displays the first found value that is read from, or written to, memory, within that range. The range you specify can be either:

- *data_low..data_high*, such as 1..10, where RealView Debugger locates the first occurrence of any data value within the range 1 to 10 being read from, or written to, memory.
- *data_low..+len*, such as 40..+10, where RealView Debugger locates the first occurrence of any data value within the range 40 to 50 being read from, or written to, memory. The *len* of 10 represents the offset value.

Find Symbol Name Match...

Enables you to locate a specific symbol-name string, such as a function name or variable, within the **Symbolic** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-33.

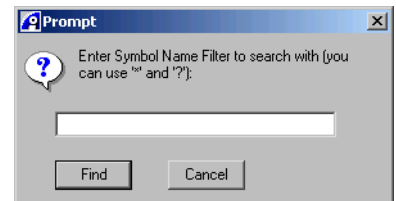


Figure 2-33 Finding a symbol name match

Enter the symbol-name filter, then click **Find**. The symbol name filter can contain the following characters:

- * A multi-character wildcard.
- ? A single-character wildcard.

For example, to find the variable `Ptr_1_Glob`, you might use `Ptr_1_*`. Alternatively, you might use `Ptr_?_Glob`.

RealView Debugger displays the first found symbol name that matches your entry.

Note

This option is available only for the **Profile** tab.

Find Next Match

Locates the next instance, within the trace output, of the search item you have last specified using any of the **Find...** options.

Find Previous Match

Searches upward within the trace output for the previous instance of the search item you have last specified using any of the **Find...** options.

2.10.9 Filtering captured information

This section describes the **Filter** menu options you can use to filter the results of a trace capture that has already been performed. This is useful if you want to refine your area of interest within the display. You can set an unlimited number of filters.

The complete menu options are:

- *Filter on Position Match...* on page 2-119
- *Filter on Time Match...* on page 2-120
- *Filter on Raw Address Match...* on page 2-121
- *Filter on Address Expression Match...* on page 2-122
- *Filter on Data Value Match...* on page 2-123
- *Filter on Symbol Name Match...* on page 2-124
- *Filter on Access Type Match...* on page 2-124
- *Filter on Percent Time Match...* on page 2-125
- *AND Filters (vs. OR)* on page 2-126
- *Clear Filtering* on page 2-126.

Filter on Position Match...

This option enables you to filter the information down to a specific element number, or range of numbers, within the trace buffer (see the description of the **Elem** column in *Columns* on page 2-74). When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-34.

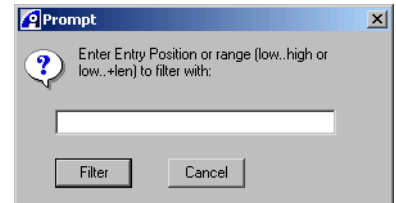


Figure 2-34 Filtering on a position match

Enter one of the following types of information, then click **Filter**:

Entry position

Specify an element number, in decimal or hexadecimal format, if you want to filter the information to display only that row.

Entry position range

Specify a range of element numbers, where RealView Debugger filters the information down to only rows within that range. The range you specify can be either:

- *low..high*, such as 1..10, where RealView Debugger displays only those rows containing Elem values within the range 1 to 10.
- *low..+len*, such as 40..+10, where RealView Debugger displays only those rows containing Elem values within the range 40 to 50. The *len* of 10 represents the offset value. You can also enter a negative value range, such as -10..10

Filter on Time Match...

Enables you to filter the information down to a specified timestamp value, or range of timestamp values, as contained in the **Time/unit** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-35.

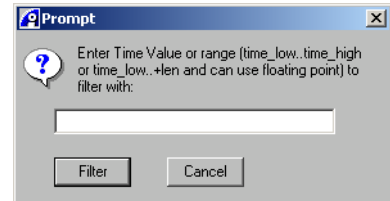


Figure 2-35 Filtering on a time match

Enter one of the following types of information, then click **Filter**:

Time value Specify a timestamp value, in the format currently used for the **Time/unit** column, if you want to filter the information to display only that row. See *Scale Time Units...* on page 2-111 for information on changing the format in which time information is displayed in the Analysis window.

Time range Specify a range of timestamp values, where RealView Debugger filters the information down to only rows within that range. (See *Scale Time Units...* on page 2-111 for information on changing the format in which time information is displayed in the Analysis window.) The range you specify can be one of:

- *time_low..time_high*, such as -100..-50, where RealView Debugger displays only those rows containing timestamp values within the range -100 to -50.
- *time_low..+len*, such as -100..+10, where RealView Debugger displays only those rows containing timestamp values within the range -100 to +10. The *len* of 10 represents the offset value.

Note

- You can use floating point values, such as -90.5.
 - Depending on your system solution, the **Time/unit** column might contain cycle numbers instead of timestamp values. In this case, you can perform filtering using cycle numbers.
-

Filter on Raw Address Match...

Enables you to filter the information down to a specified address or range of addresses, as contained in the **Address** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-36.

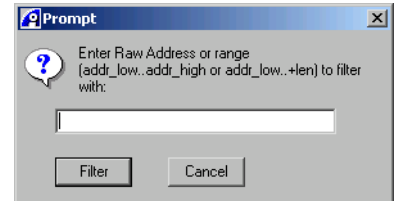


Figure 2-36 Filtering on a raw address match

Enter one of the following types of information, then click **Filter**:

Address value

Specify an address value, in decimal or hexadecimal format, if you want to filter the information to display only that row.

Address range

Specify a range of address values, where RealView Debugger filters the information down to only rows within that range. The range you specify can be either:

- *addr_low..addr_high*, such as 0x00008E50..0x0000926C, where RealView Debugger displays only those rows containing address values within that range.
- *addr_low..+len*, such as 0x00008E50..+10, where RealView Debugger displays only those rows containing address values within the range 0x00008E50 to 0x00008E5A. The *len* of 10 represents the offset value.

————— Note —————

To filter the results for an address, or range of addresses, by entering a symbolic expression, you must use the option **Filter on Address Expression Match....**

Filter on Address Expression Match...

Enables you to filter the information down to a specific address, or range of addresses, within the **Address** column, that corresponds to an expression you enter. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-37. An address expression can be a function, structure, or array symbol.

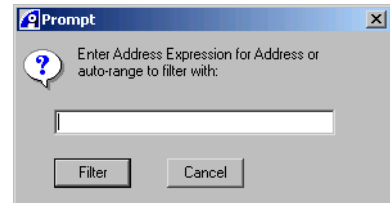


Figure 2-37 Filtering on an address expression match

Enter one of the following types of information, then click **Filter**:

Address expression

Specify an address expression if you want to filter the information to display only that row. An address expression can be one of:

- a function name
- a structure name
- an array symbol.

Auto-range

An auto-range of address values is generated from an expression that you enter, which can be any of:

- A function name, where the generated address range is from the start-to-end of the function.
- A structure, where the generated address range is from the start-to-end of the structure.
- An array symbol, where the generated address range is from the start of the variable to the end, where the end is the *start+sizeof(var)*. For example, if the start value of where the array is stored in memory is 0x8000, and the array size is 16 bytes, the end address is considered to be 0x8010 (that is, 0x8000+16).

RealView Debugger filters the information down to only rows represented by the generated auto-range.

Filter on Data Value Match...

Enables you to filter the information down to a specified data value, or range of data values, that is read from, or written to, memory. These values can be found in either of the **Data** columns (decimal or hexadecimal). When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-38.

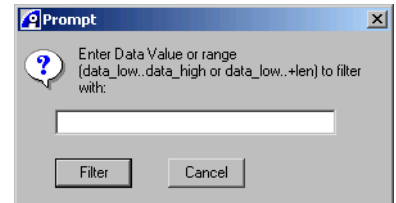


Figure 2-38 Filtering on a data value match

Enter one of the following types of information, then click **Filter**:

Data value

Specify a data value, in decimal or hexadecimal format, if you want to filter the information to display only that row.

Data range

Specify a range of data values, where RealView Debugger filters the information down to only rows of data values that are read from, or written to, memory, within that range. The range you specify can be either:

- *data_low..data_high*, such as 1..10, where RealView Debugger displays only those rows containing data values that are read from, or written to, memory within the range 1 to 10.
- *data_low..+len*, such as 40..+10, where RealView Debugger displays only those rows containing data values that are read from, or written to, memory, within the range 40 to 50. The *len* of 10 represents the offset value.

Filter on Symbol Name Match...

Enables you to filter the information down to a specified symbol-name string (such as a function name or variable) or range of symbol-name strings, as contained in the **Symbol** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-39.

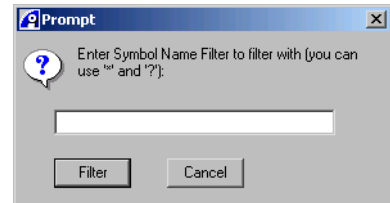


Figure 2-39 Filtering on a symbol name match

Enter the symbol name filter, then click **Filter**.

The symbol name filter can contain the following characters:

- * A multi-character wildcard.
- ? A single-character wildcard.

RealView Debugger displays only those rows containing the symbol name you specify.

For example, to display only the rows containing the variable `Ptr_1_Glob`, you might use `Ptr_1_*`. Alternatively, you might use `Ptr_?_Glob`.

Filter on Access Type Match...

Enables you to filter the information down to one or more selected access types, as contained in the **Type** column. When you select this option, a List Selection dialog box is displayed, as shown in Figure 2-40.

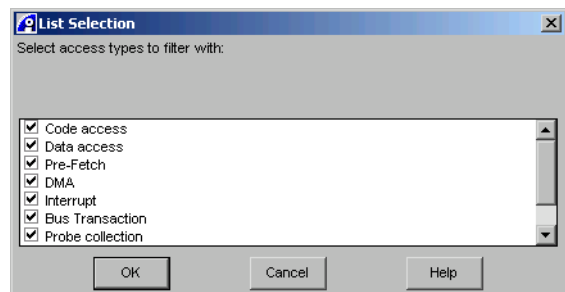


Figure 2-40 Filtering on an access type match

Select one or more access types to be included in the filtering operation and click **OK**. The following access types are available:

- Code access
- Data access
- Pre-Fetch
- DMA (Direct Memory Access)
- Interrupt
- Bus Transaction
- Probe collection
- Pin/Signal change
- Errors (non-trace).

Note

The access types shown in the List Selection dialog box are those supported by RealView Debugger, but not all of the options listed are supported for tracing by all targets. For example, the prefetch, DMA, Bus Transaction, Probe collection, and Pin/Signal change options are not supported for ETM targets.

If you filter on an unsupported option, the filtering is performed, but no matching data is found, and the Analysis window displays no information. To return the trace data to the Analysis window, you must clear the filter using the **Clear Filtering** option.

You can also perform Access Type filtering using the Rows entries in the **Trace Data** menu.

Filter on Percent Time Match...

Used only in the **Profile** tab, this option enables you to filter the information down to a specified percentage of execution time, as displayed in the **Exec%** column. When you select this option, a Prompt dialog box is displayed, as shown in Figure 2-41.

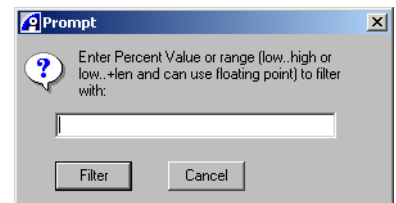


Figure 2-41 Filtering on a percent time match

Enter any of the following, then click **Filter**:

Percent value

Specify a percent time value. For example, enter 50 if you want to filter the information to display only the row(s) representing the percentage-of-execution value of 50% or greater.

Percent range

Specify a *low*..*high* range of percent values, such as 50..*60*, where RealView Debugger displays only those rows containing percentage-of-execution values within the range 50% to 60%.

Percent range with offset

Specify a *low*..*+len* range, such as 40..*+10*, where RealView Debugger displays only those rows containing percentage-of-execution values within the range 40% to 50%. The *len* of 10 represents the offset value.

———— Note ————

You can also use floating point values such as 40.5.

AND Filters (vs. OR)

Changes the way in which the filters are used together. You can set an AND or OR condition. Select this option to set an AND condition, and deselect it for an OR condition (the default).

For example, if you choose to configure a **Filter on Position Match...** and a **Filter on Data Value Match...**, the following would apply, depending on whether the **AND Filters (vs. Or)** option is selected:

- if selected (AND), the filtering process returns trace information for only the areas of execution where both the position and data value match criteria you have entered are satisfied
- if deselected (OR), the filtering process returns trace information for the areas of execution where either the position or data value match criterion you have entered is satisfied.

Clear Filtering

This option clears any filters that you have set up on the results of a trace capture that has already been performed.

2.10.10 Saving and loading trace information

This section describes how you can use the **File** menu of the Analysis window to store and retrieve captured trace and profiling information. The complete menu options are as follows:

- *Load Trace Buffer from File...*
- *Save Trace Buffer to File...*
- *Save Filtered Trace Buffer to File...* on page 2-128
- *Close Loaded File* on page 2-128
- *Print Trace Lines...* on page 2-128
- *Connection* on page 2-129
- *Exit Window* on page 2-129.

Load Trace Buffer from File...

This option enables you to load a previously saved trace buffer from a file, which you can re-analyze. This option is useful in cases where you are performing a trace capture that takes a long time to reach the point of interest, and you do not want to have to repeat the process. You can also analyze the profiling information of the saved trace buffer even after you continue to make modifications to the source code.

Note

For details on the different file types you can load, see the description of the option **Save Trace Buffer to File**.

Save Trace Buffer to File...

This option enables you to save the current trace information to a file. When you select this option, you are prompted to select one of the following options from the List Selection dialog box:

Text file containing display lines

Stores a tabulated text file, with the extension `.txt`, containing what is displayed in the current tab view of the Analysis window. This file type cannot be reloaded into the Analysis window.

Full dump of Trace contents

Stores a binary file, with the extension `.trc`, containing the complete information that RealView Debugger uses to generate the trace information, including any profiling information.

Minimal dump of Trace contents (timing+address+type)

Stores a binary file, with the extension `.trm`, containing only the timing, address, and type information that RealView Debugger uses to generate the trace information, including any profiling information. When you load this file in the future, RealView Debugger reconstructs the full trace information from these three attributes.

Warning

If you load a file of this type in a future trace session, the data values present at the memory locations might be different from those present when you originally saved this file, and errors and warnings are not stored.

Profiling data

Stores a binary file, with the extension `.trp`, containing only the profiling information that RealView Debugger uses. Unlike the **Minimal dump of Trace contents** option, RealView Debugger cannot reconstruct full trace information based on the contents of this file. However, if you want to save only profiling information, it is recommended you use this file type because it takes up significantly less space than a `.trc` file.

In each case, a Select Trace File to Save to dialog box is displayed, where you must specify a filename and directory. The default filename extension is dependent on the file type you have selected.

Save Filtered Trace Buffer to File...

This option is the same as **Save Trace Buffer to File...**, except that, if you have performed any filtering using the options in the **Filter** menu, this option ensures that you store the post-filtered trace information.

Close Loaded File

Closes the file that is currently loaded in the Analysis window, and clears the trace information from the window.

Print Trace Lines...

Enables you to print the trace-buffer contents contained in the Analysis window. When you select this option, a standard Print dialog box is displayed.

Connection

Enables you to attach a window to a connection, and to select a connection.

Attach Window to a Connection

Toggle this menu option on or off to control the attachment of the current Code window. When the window is unattached, this option is unchecked.

Active connections list

This part of the menu displays a list of active connections, in the order in which they were established. The current connection is marked with an asterisk *.

Exit Window

This option closes the Analysis window.

2.11 Mapping Analysis window options to CLI commands and qualifiers

This section shows the equivalent CLI commands and qualifiers to use for various Analysis window menu options. It contains the following sections:

- *File menu options*
- *Edit menu options* on page 2-131
- *View menu options* on page 2-132
- *Find menu options* on page 2-133
- *Filter menu options* on page 2-133.

———— **Note** —————

When tracing with CLI commands using the headless debugger, the trace output is not displayed on the screen. Therefore, you must save the trace buffer to a file. You can perform find and filter operations on the trace buffer using CLI commands. However, if you want to analyze the trace output, you must start RealView Debugger in GUI mode, and load the trace buffer file into the Analysis window. For more details about the CLI commands, see the *RealView Developer Kit v2.2 Command Line Reference Guide*.

2.11.1 File menu options

Table 2-7 shows the **File** menu options and the equivalent qualifiers to use with the TRACEBUFFER command to perform the same function in the CLI.

Table 2-7 File menu to TRACEBUFFER command qualifier mapping

File menu option	Command qualifier
Load Trace Buffer from File...	loadfile
Save Trace Buffer to File..., then one of the following options from the List Selection dialog box:	
• Text file containing display lines	savefile,ascii
• Full dump of trace contents.	savefile,full
• Minimal dump of trace contents.	savefile,minimal
• Profiling data.	savefile,profile

Table 2-7 File menu to TRACEBUFFER command qualifier mapping (continued)

File menu option	Command qualifier
Save Filtered Trace Buffer to File..., then one of the following options from the List Selection dialog box:	
• Text file containing display lines	savefile,ascii,filtered
• Full dump of trace contents.	savefile,full,filtered
• Minimal dump of trace contents.	savefile,minimal,filtered
• Profiling data.	savefile,profile,filtered
Close Loaded File	closefile

2.11.2 Edit menu options

Table 2-8 shows the **Edit** menu options and the equivalent commands and qualifiers to use to perform the same function in the CLI.

Table 2-8 Edit menu to ANALYZER command qualifier mapping

Edit menu option	Command qualifier
Connect Analyzer...	ANALYZER ,connect
Disconnect Analyzer	ANALYZER ,disconnect
Tracing Enabled (checked)	ANALYZER ,enable
Tracing Enabled (unchecked)	ANALYZER ,disable
Configure Analyzer Properties...	ETM_CONFIG ^a
Set Trace Buffer Size...	ANALYZER ,set_size:(n)
Store Control-flow Changes Only (checked)	ANALYZER ,collect_flow
Store Control-flow Changes Only (unchecked)	ANALYZER ,collect_all
Buffer Full Mode → Stop Processor on Buffer Full	ANALYZER ,full_stop
Buffer Full Mode → Stop Collecting on Buffer Full	ANALYZER ,full_ignore
Buffer Full Mode → Continue Collecting on Buffer Full	ANALYZER ,full_ring
Trigger Mode → Collect Trace Before Trigger	ANALYZER ,before
Trigger Mode → Collect Trace Around Trigger	ANALYZER ,around
Trigger Mode → Collect Trace After Trigger	ANALYZER ,after

Table 2-8 Edit menu to ANALYZER command qualifier mapping (continued)

Edit menu option	Command qualifier
Trigger Mode → Stop Processor on Trigger (checked)	ANALYZER ,stop_on_trigger
Trigger Mode → Stop Processor on Trigger (unchecked)	ANALYZER ,continue_on_trigger
Data Tracing Mode → Address Only	ANALYZER ,addronly
Data Tracing Mode → Data Only	ANALYZER ,dataonly
Data Tracing Mode → Data and Address	ANALYZER ,fulltrace
Automatic Tracing Mode → Off (Use tracepoints)	ANALYZER ,auto_off
Automatic Tracing Mode → Instructions Only	ANALYZER ,auto_instronly
Automatic Tracing Mode → Data Only (ETMv3 only)	ANALYZER ,auto_dataonly
Automatic Tracing Mode → Instructions and Data	ANALYZER ,auto_both
Set/Edit Event Triggers...	ANALYZER ,triggers
Clear All Event Triggers	ANALYZER ,clear_triggers
Physical to Logical Address Mapping...	ANALYZER ,map_log_phys

- a. See *Equivalent CLI command qualifiers* on page 2-27 for details about the ETM_CONFIG command qualifiers.

2.11.3 View menu options

Table 2-9 shows the **View** menu options and the equivalent commands and qualifiers to use to perform the same function in the CLI.

Table 2-9 View menu to command qualifier mapping

View menu option	Command qualifier
Update	TRACEBUFFER ,refresh
Clear Trace Buffer	ANALYZER ,clear
Show Position Relative to Trigger (checked)	TRACEBUFFER ,pos_relative
Show Position Relative to Trigger (unchecked)	TRACEBUFFER ,pos_absolute
Scale Time Units...	TRACEBUFFER ,scaletime= <i>n</i>
Define Processor Speed for Scaling...	TRACEBUFFER ,speed= <i>n</i>

2.11.4 Find menu options

Table 2-10 shows the **Find** menu options and the equivalent qualifiers to use with the TRACEBUFFER command to perform the same function in the CLI.

Table 2-10 Find menu to TRACEBUFFER command qualifier mapping

Find menu option	Command qualifier
Find Trigger	find_trigger
Find Position Match...	find_position
Find Time Match...	find_time
Find Raw Address Match...	find_address
Find Data Value Match...	find_data
Find Symbol Name Match...	find_name

2.11.5 Filter menu options

Table 2-11 shows the **Filter** menu options and the equivalent qualifiers to use with the TRACEBUFFER command to perform the same function in the CLI.

Table 2-11 Filter menu to TRACEBUFFER command qualifier mapping

Filter menu option	Command qualifier
Filter on Position Match...	posfilter
Filter on Time Match...	timefilter
Filter on Raw Address Match...	addrfilter addressfilter
Filter on Data Value Match...	dvafilter datavaluefilter
Filter on Symbol Name Match...	namefilter
Filter on Access Type Match...	typefilter accesstypefilter
Filter on Percent Time Match...	percentfilter
AND Filters (vs. OR) (checked)	and_filter
AND Filters (vs. OR) (unchecked)	or_filter
Clear Filtering	clearfilter

2.12 Examples of using trace in RealView Debugger

This section provides examples of how you can use the tracing features of RealView Debugger to solve typical development problems and analyze certain elements of the execution of your application. It contains the following sections:

- *About the examples*
- *Finding the cause of a data abort* on page 2-135
- *Capturing profiling information* on page 2-143
- *Setting up a conditional tracepoint* on page 2-149.

2.12.1 About the examples

It is recommended that you use these examples before using trace on your own application images, because they are designed to introduce you to the trace features of RealView Debugger, and do not assume that you have any experience of using RealView Debugger.

These examples require you to use the following images:

- `install_directory\RVDK\Examples\2.2\build_number\windows\primes\DebugRe1\primes.axf`
- `install_directory\RVDK\Examples\2.2\build_number\windows\dhrystone\DebugRe1\dhrystone.axf`.

The source file components are located in the main project directories `...\primes` and `...\dhrystone`.

These examples require you to have the following RealView Debugger-supported trace hardware components, which must be installed and connected properly:

- a JTAG interface unit, such as RealView ICE
- trace capture hardware, such as RealView Trace
- an ETM-enabled ARM processor.

See Appendix A *Setting up the Trace Hardware* and Appendix B *Setting up the Trace Software* for information on configuring your system.

Before beginning each example:

- start RealView Debugger
- ensure that your trace hardware and target are properly configured
- connect to a target.

The following examples are provided:

- *Finding the cause of a data abort* on page 2-135
- *Capturing profiling information* on page 2-143

- *Setting up a conditional tracepoint* on page 2-149.

2.12.2 Finding the cause of a data abort

This example demonstrates how you can use the trace features of RealView Debugger to locate a problematic area in your application. It assumes that you are using an ETM-based target, for example an ARM966E-S processor.

The `primes.axf` image used in this example is designed to calculate the n th prime number, where you are prompted to indicate n when running the image. However, execution of the image results in a data abort.

This example is in two parts:

1. *Procedure for finding the cause of a data abort*
2. *Displaying inferred registers* on page 2-141.

Procedure for finding the cause of a data abort

To determine the cause of the data abort in the Primes example:

1. Connect to your debug target.
2. Load the example image `primes.axf` into the debugger. This file is located in the `install_directory\RVDK\Examples\...\primes` directory. The tab for `primes.cpp` is displayed in the File Editor pane.
3. Disable the Data Abort vector catch to prevent any Data Abort exception from being caught by the debugger. How you do this depends on your JTAG interface unit:

RealView ICE interface unit

If you are using RealView ICE:

1. Select **Debug** → **Processor Exceptions...** from the Code window main menu to open the Processor Exceptions List Selection dialog box.
2. Make sure that required exception is not selected in the list of processor events.
3. Click **OK** to close the dialog box.

Alternatively, you can disable the Data Abort vector catch using the `BGLOBAL` command (see the *RealView Developer Kit v2.2 Command Line Reference Guide* for more details). For example:

```
bglobal,disable "data abort"
```

Non-ARM JTAG interface unit

If you are using a non-ARM JTAG interface unit, see the documentation accompanying your hardware for information on how to disable the vector catch for an exception.


4. Select **View** → **Analysis Window** from the Code window main menu to display the Analysis window.
5. Configure the general tracing options. From the Analysis window main menu:
 - a. Select **Edit** → **Trigger Mode** → **Collect Trace Before Trigger**.
 - b. Select **Edit** → **Trigger Mode** → **Stop Processor On Trigger**.
 - c. Select **Edit** → **Data Tracing Mode** → **Data and Address**.

This ensures that a buffer-load of trace data is captured for the area of your application occurring before any trigger point you set. This is useful when you want to see the events leading up to, but not occurring after, the trigger point. In this example, the trigger point represents the area in the application where the data abort occurs.

Note

You can use the default ETM configuration for the target.

6. Display the Data Abort vector as follows:
 - a. In the File Editor pane of the Code window, select the **Dsm** tab to display the disassembly of the application image.
 - b. Right-click in the white space to the right of the disassembly code in the File Editor pane. A context menu is displayed.
 - c. Select **View from Location...** from the context menu. A Prompt dialog box is displayed.
 - d. In the dialog box, enter the value 0 and click **Set**. This displays the region of memory containing the exception vectors. An arrow ➡ is placed next to the address 0x00000000.
7. Set a trace trigger point on the Data Abort vector as follows:
 - a. In the File Editor pane of the Code window, select the **Dsm** tab to display the disassembly of the application image.
 - b. Right-click in the left margin at the address 0x00000010 to display the context menu.
 - c. Select **Set/Toggle Tracepoint...** from the context menu. A List Selection dialog box is displayed.

- d. In the List Selection dialog box, select **Set Trigger** and click **OK**. Another arrow  is placed next to the address 0x10. This arrow indicates the trigger point you have set, and details of this trigger tracepoint are displayed in the Break/Tracepoints pane of the Code window, as shown in Figure 2-42.

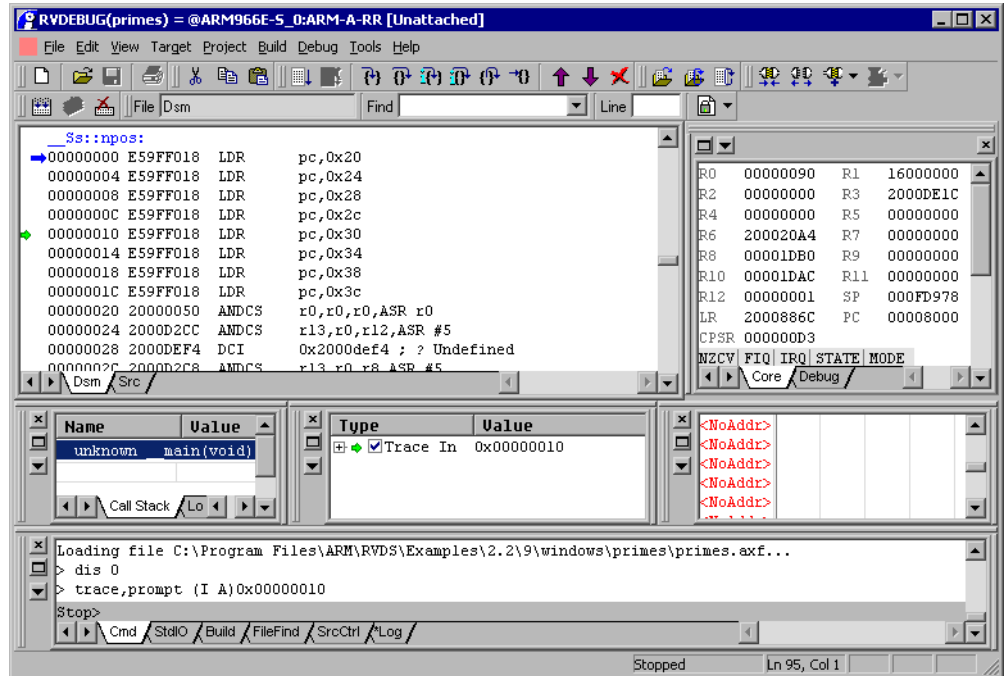



Figure 2-42 Setting a trigger point

8. Set a start of trace range tracepoint to capture instruction and data accesses as follows:
 - a. Right-click in the left margin at the address 0x0 to display the context menu.
 - b. Select **Set/Toggle Tracepoint...** from the context menu. A List Selection dialog box is displayed.
 - c. In the List Selection dialog box, select **Start of Trace Range (Instruction and Data)** and click **OK**. Another arrow  is placed next to the address 0x0. This arrow indicates the start of the trace range you have set, and details of this tracepoint are displayed in the Break/Tracepoints pane of the Code window, as shown in Figure 2-43 on page 2-138. The end of the trace range is automatically set to the end of memory space (0xFFFFFFFF).

Note

If you do not set any tracepoints (that is, the trigger and trace range in this example), then Automatic Tracing Mode is used. See *Automatic Tracing Mode* on page 2-108 for information on selecting the Automatic Tracing Mode.

If you set a trigger but no trace range, then all instructions are traced, but no data (see *Trigger* on page 2-29).

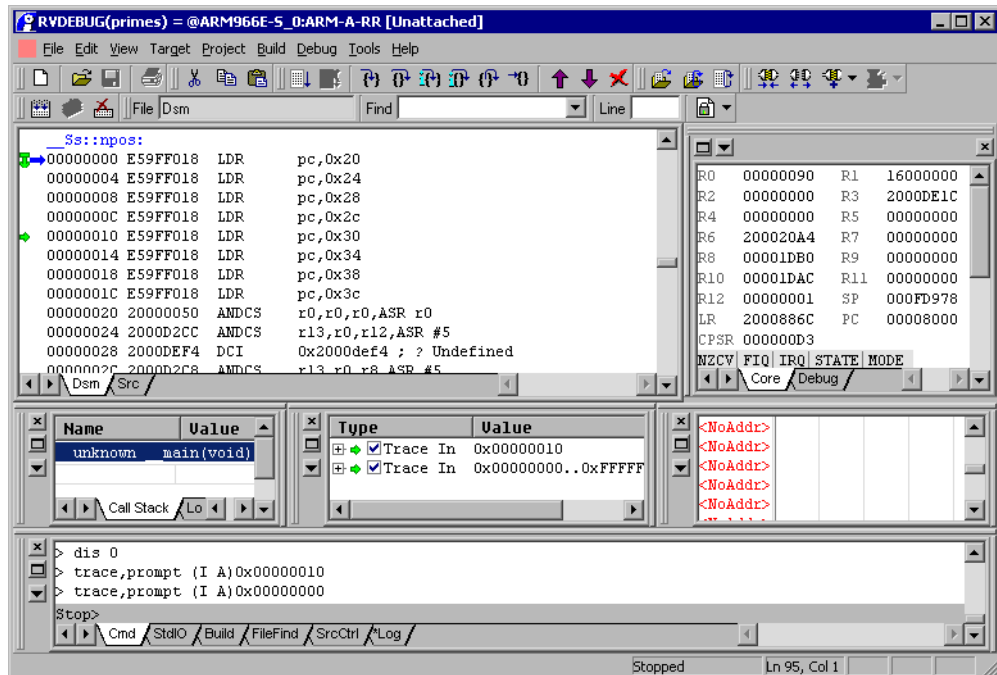


Figure 2-43 Setting a trace range

9. Select **Debug** → **Run** from the Code window main menu. The image is executed and a prompt is displayed in the Output pane at the bottom of the Code window.
10. In the Output pane (ensuring that the **StdIO** tab is selected), enter 20 and press Enter. As a result of a deliberate error in the image, a data abort occurs. Because you have set a trigger point at the data abort address, the results of the trace capture are returned.
11. To view the results of the trace capture, select **View** → **Analysis Window** from the Code window main menu. The Analysis window is opened, and the **Trace** tab is displayed. The disassembly of program instructions is displayed by default, as shown in Figure 2-44 on page 2-139.

The screenshot shows the RealView Debugger's Trace tab. The window title is 'Analysis(primes) = @ARM966E-S_0:ARM-A-RR [Unattached]'. The status bar indicates 'Tracing enabled'. The main table displays the following columns: Elen, Time/pico, Type, Symbolic, Address, Opcode, and Other. The table contains multiple rows of execution data, including instructions like 'main', 'scatterload_rt2', 'region_table', and 'scatterload_null'. A warning 'Warning: Trace pause' is visible at the top of the table. The bottom of the window shows a search bar with the text '[0x56974,0x64E,56974] <No line selected>' and tabs for 'Trace', 'Source', and 'Profile'.

Elen	Time/pico	Type	Symbolic	Address	Opcode	Other
-354558			Warning: Trace pause			
-354545	-2.56985e+	Exec	main	0x00008000	0xEA000000 B	__scatte
-354541	-2.56985e+	Exec	scatterload_rt2	0x00008008	0xE28FC028 ADR	r12,(pc):
-354531	-2.56985e+	Exec	scatterload_rt2	0x0000800C	0xE89C0C00 LDMIA	r12,{r10
-354531	-2.56985e+	R Data	region_table	0x00008038	<Data>	0x88 0xE9
-354531	-2.56985e+	R Data			<Data>	0x98 0xE9
-354527	-2.56985e+	Exec	scatterload_rt2	0x00008010	0xE08AA00C ADD	r10,r10,
-354523	-2.56985e+	Exec	scatterload_rt2	0x00008014	0xE24A7001 SUB	r7,r10,#
-354519	-2.56985e+	Exec	scatterload_rt2	0x00008018	0xE08BB00C ADD	r11,r11,
-354515	-2.56985e+	Exec	scatterload_null	0x0000801C	0xE15A000B CMP	r10,r11
-354511	-2.56985e+	NoExec	scatterload_null	0x00008020	0x0A000739 BEQ	__rt_ent
-354497	-2.56985e+	Exec	scatterload_null	0x00008024	0xE8BA000F LDMIA	r10!,{r0
-354497	-2.56985e+	R Data		0x000169C0	<Data>	't' 'j' 0
-354497	-2.56985e+	R Data			<Data>	't' 'j' 0
-354497	-2.56985e+	R Data			<Data>	0xA0 0xE
-354497	-2.56985e+	R Data			<Data>	'0' 0x80
-354493	-2.56985e+	Exec	scatterload_null	0x00008028	0xE24FE014 ADR	r14,(pc)
-354489	-2.56985e+	Exec	scatterload_null	0x0000802C	0xE3130001 TST	r3,#1
-354485	-2.56985e+	NoExec	scatterload_null	0x00008030	0x1047F003 SUBNE	pc,r7,r3
-354484	-2.56985e+	Exec	scatterload_null	0x00008034	0xE1A0F003 MOV	pc,r3
-354469	-2.56985e+	Exec	scatterload_zeroinit	0x00008040	0xE3A04000 MOV	r4,#0
-354465	-2.56985e+	Exec	scatterload_zeroinit	0x00008044	0xE3A05000 MOV	r5,#0
-354461	-2.56985e+	Exec	scatterload_zeroinit	0x00008048	0xE3A06000 MOV	r6,#0
-354457	-2.56985e+	Exec	scatterload_zeroinit	0x0000804C	0xE3A0C000 MOV	r12,#0
-354453	-2.56985e+	Exec	scatterload_zeroinit	0x00008050	0xE2522010 SUBS	r2,r2,#0
-354441	-2.56985e+	Exec	scatterload_zeroinit	0x00008054	0x28A11070 STMCSIA	r1!,{r4-
-354441	-2.56985e+	W Data	stdin	0x00016A74	<Data>	'\0' '\0'
-354441	-2.56985e+	W Data			<Data>	'\0' '\0'

Figure 2-44 Results displayed in the Trace tab

12. Hide the **Opcode** column and view the interleaved source boundaries to help you locate the error in `CalculatePrimes()`.

From the Analysis window main menu, do the following:

- a. Select **Trace Data** → **Opcode** (see *Columns in the Trace tab* on page 2-80 for more details).
- b. Select **Trace Data** → **Interleaved Source** (see *Rows in the Trace tab* on page 2-83 for more details).
- c. Select **Trace Data** → **Function Boundaries** (see *Rows in the Trace tab* on page 2-83 for more details).
- d. Select **Find** → **Find Trigger**.

RealView Debugger locates the row in the output representing the trigger point you have set, shown in Figure 2-45 on page 2-140.

Note

Some of the details shown in Figure 2-45, for example the values in the Elem column, might be different depending on your JTAG interface unit and target processor.

Elem	Time/pico	Type	Symbolic	Address	Other
-148	-1900000	NoExec	__Heap_Free	0x0000B940	MOVNE pc,r14
-140	-1800000	Exec	__Heap_Free	0x0000B944	LDR r3,[r0,#4]
-140	-1800000	R Data		0x000192A8	<Data> '\0' '\0' '\0' '\0'
-134	-1750000	Exec	__Heap_Free	0x0000B948	STR r3,[r1,#4]
-134	-1750000	W Data		0x00019220	<Data> '\0' '\0' '\0' '\0'
-126	-1650000	Exec	__Heap_Free	0x0000B94C	LDR r0,[r0,#0]
-126	-1650000	R Data		0x000192A4	<Data> 0x84 0x06 '\0' '\0'
-122	-1600000	Exec	__Heap_Free	0x0000B950	ADD r0,r2,r0
-116	-1500000	Exec	__Heap_Free	0x0000B954	STR r0,[r1,#0]
-116	-1500000	W Data		0x0001921C	<Data> '\f' '\a' '\0' '\0'
-115	-1500000	Exec	__Heap_Free	0x0000B958	MOV pc,r14
-96	-1250000	Exec	_ZdlPv	0x00011134	LDMFD r13!,(r12,pc)
-96	-1250000	R Data		0x0001FFA8	<Data> 0xD8 0xFF 0x01 '\0'
-96	-1250000	R Data			<Data> 0x98 'J' 0x01 '\0'
-80		Exec	std::vector<int, std::a	0x00014A98	MOV r0,r4
-72		Exec	std::vector<int, std::a	0x00014A9C	LDMFD r13!,(r4,pc)
-72	-950000	R Data		0x0001FFB0	<Data> 0xFF 0xFF 0xFF 0xFF
-72	-950000	R Data			<Data> 0x1C 0x86 '\0' '\0'
PRIMES	149		OutputResults(*answer, prime, Time);		
-56	-750000	Exec	main\#149	0x0000861C	ADD r3,r13,#0x20
-46	-650000	Exec	main\~#149	0x00008620	LDMIA r3,[r2,r3]
-46	-650000	R Data		0x0001FFD8	<Data> '(' 0x14 0xAE 'G'
-46	-650000	R Data			<Data> 0xE1 'z' 0x94 '?'
-14	-200000	Exec		0x00008624	DATA ABORT
-14	-200000	R Data		0xFFFFFFFF	<Data> '\0' '\0' '\0' '\0'
*4	50000	Exec		0x00000010	DATA ABORT

Figure 2-45 Interleaved source in Trace tab

Observations on tracing the data abort

The red line labeled DATA ABORT, shown in Figure 2-45, is the trigger point (at address 0x00000010). In the rows with an **Elem** value of -72 to -46, you can see that a value of -1 (0xFFFFFFFF) is being used as a pointer. When OutputResults() attempts to use the value on input, the Data Abort occurs (the rows with an **Elem** value of -14).

The error is in the way the CalculatePrimes() function returns the prime number calculated. The main() function expects an address to be returned, which it can place in a pointer, and then use for outputting. However, the CalculatePrimes() function is passing a number not an address. This means that the image tries to access memory location -1 (0xFFFFFFFF).

In this example, the vector is initialized to one space more than is required, and all elements are set to -1. This ensures that the calculation function passes back the one unused element.

Displaying inferred registers

RealView Debugger enables you to view the values that the registers held at each instruction in the trace output. The values are obtained by examining line of the trace output, so not all register values can be inferred. The registers are interleaved with the trace output, and the instruction boundaries are also displayed.










To display inferred registers for the data abort example:

1. If you have not already done so, perform the procedure in *Procedure for finding the cause of a data abort* on page 2-135.
2. From the Analysis window menu:
 - a. Select **Trace Data** → **Inferred Registers** (see *Rows in the Trace tab* on page 2-83 for more details).
 - b. Select **Find** → **Find Trigger**.

The **Trace** tab looks like that shown in Figure 2-46 on page 2-142. You might have to scroll down to view the information following the trigger point.

Analysis(primes) = @ARM966E-S_0:ARM-A-RR [Unattached]

File Edit View Find Filter Sort Trace Data Profiling Data Help



Status: Tracing enabled

Elem	Time/pico	Type	Symbolic	Address	Other										
R0	0001FFE4	R1	0001921C	R2	00000088	R3	00000000	R4	FFFFFFFF	R5		R6		R7	
R8		R9		R10		R11		R12	0001FFD8	SP	0001FFB8	LR	00011134	PC	0000861C
NZCV 0110 STATE ARM															
-72	-1000000		R Data					0x0001FFB0	<Data>	0xFF 0xFF 0xFF 0xFF					
-72	-1000000		R Data						<Data>	0x1C 0x86 '\0' '\0'					
PRIMES 150															
-56	-800000	Exec	main\#147..#149					0x0000861C	ADD	r3,r13,#0x20					
R0	0001FFE4	R1	0001921C	R2	00000088	R3	0001FFD8	R4	FFFFFFFF	R5		R6		R7	
R8		R9		R10		R11		R12	0001FFD8	SP	0001FFB8	LR	00011134	PC	00008620
NZCV 0110 STATE ARM															
-46	-650000	Exec	main\#149					0x00008620	LDMIA	r3,{r2,r3}					
R0	0001FFE4	R1	0001921C	R2	47AE147B	R3	3F947AE1	R4	FFFFFFFF	R5		R6		R7	
R8		R9		R10		R11		R12	0001FFD8	SP	0001FFB8	LR	00011134	PC	00008624
NZCV 0110 STATE ARM															
-46	-650000		R Data					0x0001FFD8	<Data>	'{' 0x14 0xAE 'G'					
-46	-650000		R Data						<Data>	0xE1 'z' 0x94 '?'					
-14	-200000	Exec						0x00008624	DATA ABORT						
R0	0001FFE4	R1	0001921C	R2	47AE147B	R3	3F947AE1	R4	FFFFFFFF	R5		R6		R7	
R8		R9		R10		R11		R12		SP		LR		PC	00000010
NZCV --- STATE ---															
-14	-200000		R Data					0xFFFFFFFF	<Data>	'\\0' '\\0' '\\0' '\\0'					
*4 0 Exec 0x00000010 DATA ABORT															
R0		R1		R2		R3		R4		R5		R6		R7	
R8		R9		R10		R11		R12		SP		LR		PC	
NZCV --- STATE ---															
4	0		R Data					0x00000030	<Data>	0xC4 0xD2 '\\0' ' '					
[0x56954,0x649,56954] DATA ABORT															

0x56954,0x649,56954

DATA ABORT

Trace

Source

Profile

Figure 2-46 Inferred registers in the Trace tab

2.12.3 Capturing profiling information

This example demonstrates how you can use RealView Debugger to capture profiling information for your application.

The `dhrystone.axf` image used in this example performs a benchmarking sample that is executed n number of times, where you are prompted to indicate n when running the image. By performing multiple runs, you can sum the profiling data. This example assumes that you want to analyze the execution times of all functions that are executed in the main for loop that is run repeatedly. It also assumes that you are using an ETM-based target.

This example is in two parts:

1. *Procedure for capturing profiling data*
2. *Procedure for summing profiling data over multiple runs* on page 2-147.

Procedure for capturing profiling data


To capture profiling information using the `dhrystone` example:

1. Load the example image `dhrystone.axf` into the debugger. This file is located in the `install_directory\RVDK\Examples\...\dhrystone\DebugRel` directory. The tab for `dhry_1.c` is displayed in the File Editor pane.
2. Select **Edit** → **Advanced** → **Show Line Numbers** from the Code window main menu to display the line numbers in `dhry_1.c`.
3. Select **View** → **Analysis Window** from the Code window main menu to view the Analysis window, and position the window so that it does not cover the Code window.
4. Configure the ETM for profiling:
 - a. Select **Edit** → **Configure Analyzer Properties...** from the Analysis window main menu to display the Configure ETM dialog box (see *Configuring the ETM* on page 2-16 for more details on this dialog box).
 - b. To generate profiling information you must enable one of the settings that causes this to be captured. For RealView Trace, you can select either or both of the settings **Enable Timestamping** and **Cycle accurate tracing**. For this example, select **Enable Timestamping**.

————— Note —————

Be aware that selecting **Cycle accurate tracing** might give less accurate results than timestamping.

- c. Click **OK** to close the Configure ETM dialog box.

5. Set a trace start point at the start of the program loop as follows:
 - a. In the dhry_1.c source tab, scroll down the source file and right-click in the gray area to the left of the code listing in line 146 to display the context menu. This line represents the start of a for loop.
 - b. Select **Set/Toggle Tracepoint...** from the context menu. A List Selection dialog box is displayed.
 - c. Select **Trace Start Point** from the List Selection dialog box. An arrow  is placed next to line 146 to indicate the start point you have set, and details of this control point are displayed in the Break/Tracepoints pane of the Code window, as shown in Figure 2-47.

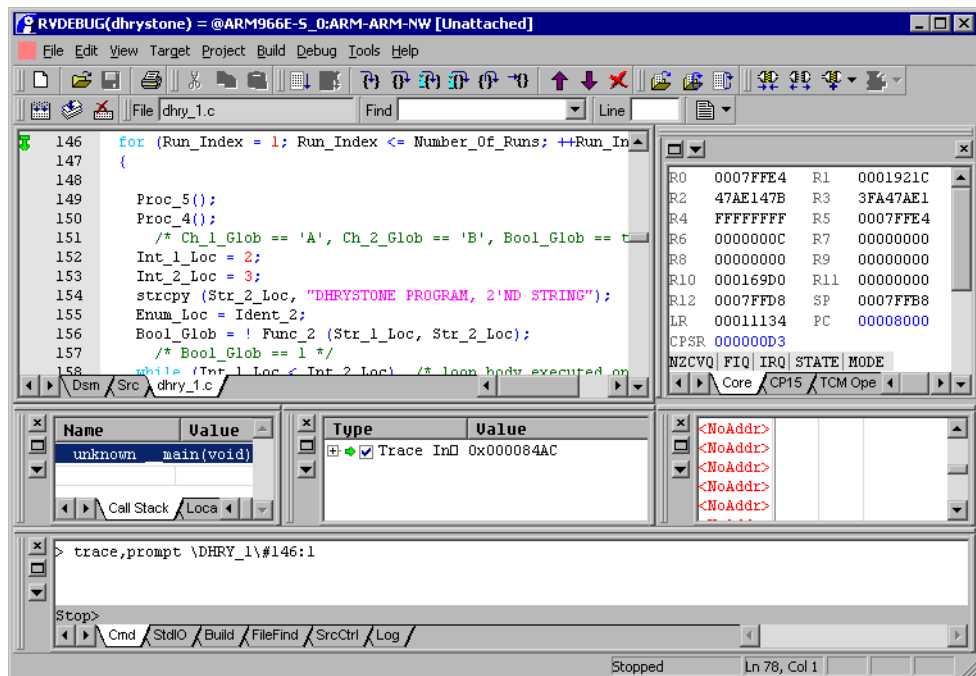



Figure 2-47 Setting a trace start point

6. Set a trace end point after the end of the program loop as follows:
 - a. Scroll down the source file and right-click in the gray area to the left of the code listing at line 204 to display the context menu.
By placing the end point after the end of the loop, you ensure that RealView Debugger captures all the iterations of the loop, rather than a single iteration.

In addition, the area bounded by the trace start and end points does not include any code that activates semihosting. The semihosting mechanism can affect the profiling results.

- b. Select **Set/Toggle Tracepoint...** from the context menu. A List Selection dialog box is displayed.
- c. Select **Trace End Point** from the List Selection dialog box. An arrow  is placed next to line 204 to indicate the end point you have set, and details of this control point are displayed in the Break/Tracepoints pane of the Code window.

Because you have set trace start and end points, and not a trace range, you are instructing RealView Debugger to capture and display all trace information between the start and end points, including any instructions that might be branched to between the points. For more details on these types of tracepoints, see *Setting unconditional tracepoints* on page 2-39.

7. Select **Debug → Run** from the Code window main menu. The image is executed and a prompt is displayed in the Output pane at the bottom of the Code window.
8. In the Output pane, enter the number of runs through the benchmark you want RealView Debugger to perform, in this case **5000**, and press Enter. Trace capture begins for the area of execution you have specified using tracepoints, that is, for the for loop area of code. The Analysis window is updated with the results of the trace capture.
9. In the Analysis window, click the **Profile** tab to display the profile data, as shown in Figure 2-48.

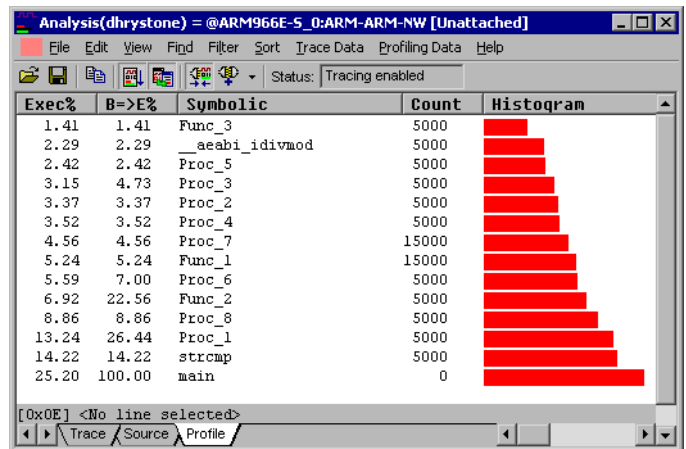


Figure 2-48 Results displayed in the Profile tab

In the **Profile** tab, as shown in Figure 2-48 on page 2-145, the execution times for all functions accessed during the for loop are displayed, and a graphical representation of their respective execution times is shown in the **Histogram** column. For details on the types of information displayed in each column, see *Columns* on page 2-74.

- 10. To view call-graph data for these results:
 - a. Select **Profiling Data** → **Parents of Function** from the Analysis window main menu.
 - b. Select **Profiling Data** → **Children of Function** from the Analysis window main menu.

The **Profile** tab displays parents and children of each function, as shown in Figure 2-49.

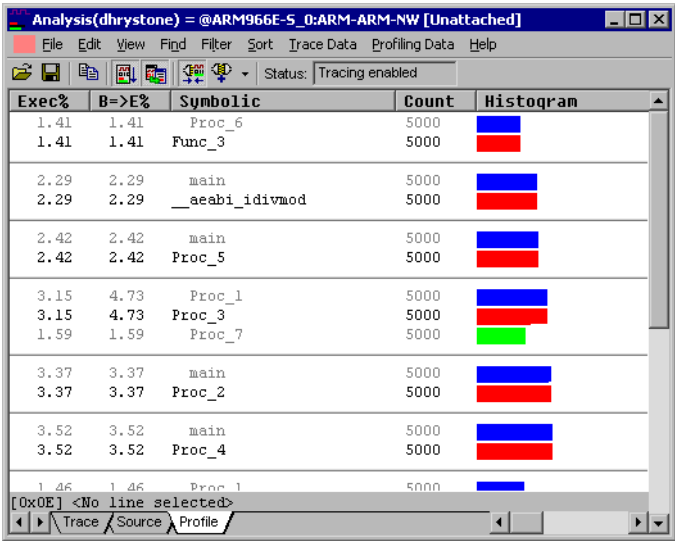


Figure 2-49 Call-graph data displayed in the Profile tab

In addition to displaying the execution times for each function accessed during the for loop, the execution times of the parents and children of these functions are also displayed. Figure 2-50 shows the data for a single function, Func_2.

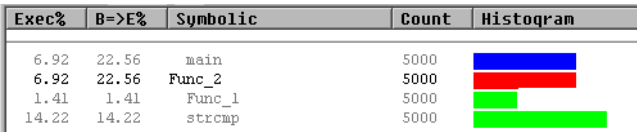


Figure 2-50 Call-graph data for Func_2

The **Exec%** column shows that:

- 6.92% of the total execution time was spent in code of the function Func_2 when called from the parent main.
- 6.92% of the total execution time was spent in code of the function Func_2.
- 1.41% of the total execution time was spent in code of the function Func_1 when called as a child from Func_2.
- 14.22% of the total execution time was spent in code of the function strcmp when called as a child from Func_2.

The **B=>E%** column shows that:

- 22.56% of the total execution time was spent in calls to the function Func_2 and its children when called from the parent main.
- 22.56% of the total execution time was spent in calls to the function Func_2 and its children.
- 1.41% of the total execution time was spent in calls to the function Func_1 called as a child from Func_2.
- 14.22% of the total execution time was spent in calls to the function strcmp called as a child from Func_2.

The **Count** column shows that:

- There were 5000 calls from the function main to the function Func_2.
- There were 5000 calls to the function Func_2.
- There were 5000 calls to the function Func_1 from the function Func_2.
- There were 5000 calls to the function strcmp from the function Func_2.

11. You can sum profiling data over additional runs of dhrystone as described in *Procedure for summing profiling data over multiple runs*.

Procedure for summing profiling data over multiple runs

To sum profiling data over multiple runs, do the following:

1. If you have not yet done so, perform the procedure described in *Procedure for capturing profiling data* on page 2-143.
2. Select **Profiling Data** → **Sum Profiling Data** from the Analysis window main menu.
3. Select **Debug** → **Set PC to Entry Point** from the Code window main menu.

Note

If you reload the image, or load a new image, the trace details are cleared and **Sum Profiling Data** is disabled.

4. Select **Debug** → **Run** from the Code window main menu.
5. Enter 500 for the number of runs through the benchmark. The profiling data is updated as shown in Figure 2-51.

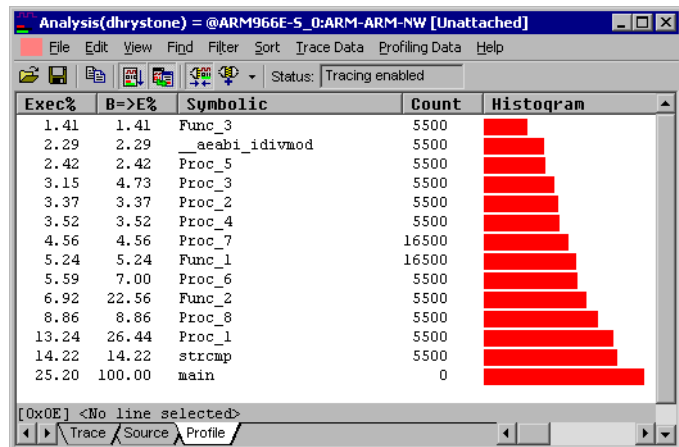


Figure 2-51 Summing profiling data over multiple runs

If you no longer want to sum profiling data, select **Profiling Data** → **Sum Profiling Data** from the Analysis window main menu to deselect it.

Observations on summing profiling data over multiple runs

Compare Figure 2-51 with Figure 2-48 on page 2-145. The summing of profiling data reveals the following information:

Count column

The **Count** column reveals the following:

- After the first run:
 - there were 15000 calls to functions Proc_7, Func_1
 - there were 5000 calls to all other functions.
- After the second run:
 - there were 16500 calls to functions Proc_7, Func_1
 - there were 5500 calls to all other functions.

B=>E% column

The **B=>E%** column reveals the following:

- The total execution time after two runs is equal to the total execution time of the first run, plus the total execution time of the second run.
- The time taken to execute a particular function after two runs is equal to the time taken to execute it after the first run.
- The percentage of the total execution time that is spent in the code of a particular function is the same as that after the first run. However, this might not always be the case.

Exec% column

The **Exec%** column reveals the following:

- The total execution time after two runs is equal to the total execution time of the first run, plus the total execution time of the second run.
- The time spent from entry to exit of a particular function after two runs is equal to the time spent from entry to exit of that function after the first run.
- The percentage of the total execution time that is spent in calls to a particular function is the same as that after the first run. However, this might not always be the case.

2.12.4 Setting up a conditional tracepoint

This example demonstrates how you can set up a conditional tracepoint (see *Setting conditional tracepoints* on page 2-48 for more details). It uses the `dhrystone.axf` image described in *Capturing profiling information* on page 2-143. This example assumes that you are connected to an ETM-based target, and that you want to trigger trace in `Proc_2`, but only after 50 executions of the function `Proc_1`.

Procedure for setting conditional tracepoints

To set up a conditional tracepoint using the `dhrystone` example:

1. Load the example image `dhrystone.axf` into the debugger. This file is located in the `install_directory\RVDK\Examples\...\dhrystone\DebugRel` directory. The tab for `dhry_1.c` is displayed in the File Editor pane.
2. Select **Debug → Tracepoints → Trace on X after Y executed N times...** from the Code window main menu. The Trace on X after Y executed N times dialog box is displayed, as shown in Figure 2-52 on page 2-150.

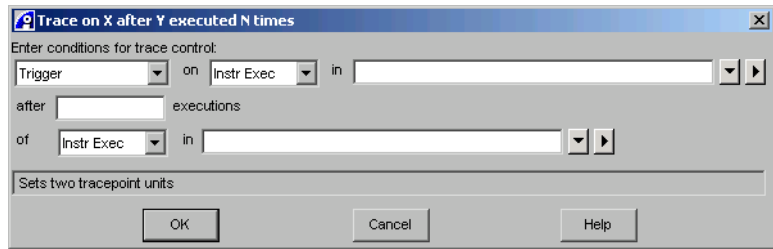


Figure 2-52 Trace on X after Y executed N times dialog box

3. Set up the tracepoint as follows:
 - a. Select **Trigger** from the first drop-down list.
 - b. Select **Instr Exec** from the second drop-down list.
 - c. Click on the drop-down arrow ▼ to the right of the text box to display a menu of items saved in your personal history file. Select **<Function List...>** from this menu to display the Function List dialog box. Select **DHRY_1\Proc_2 of @dhrystone** from the list of functions. This selects the function Proc_2 as the function to trigger on.
 - d. Type 50 into the text box on the second line of the dialog box, to specify that Proc_1 is to be executed 50 times.
 - e. Select **Instr Exec** from the drop-down list in the last line of the dialog box.
 - f. Click on the drop-down arrow ▼ to the right of the text box to display a menu of items saved in your personal history file. Select **<Function List...>** from this menu to display the Function List dialog box. Highlight **DHRY_1\Proc_1 of @dhrystone** in the list of functions, then click **Select**. This selects the function Proc_1 as the function that must be executed 50 times before the trigger can occur. The completed dialog box appears as shown in Figure 2-53.

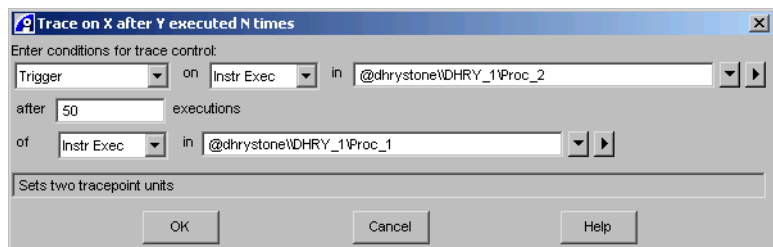


Figure 2-53 Completed conditional tracepoint dialog box

- g. Click **OK** to set the tracepoint as specified.

4. If you want to view the tracepoint details, select **View → Break/Tracepoints** from the Code window main menu to display the Break/Tracepoints pane, as shown in Figure 2-54.

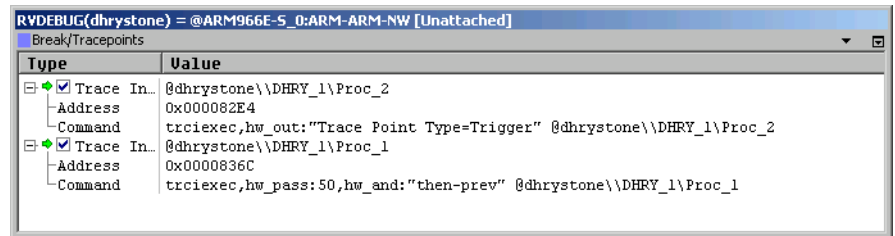


Figure 2-54 Break/Tracepoints pane with conditional tracepoints set

5. Select **View → Analysis Window** from the Code window main menu to view the Analysis window.
6. Select **Edit → Trigger Mode** from the Analysis window main menu, and ensure that **Collect Trace Before Trigger** is selected.
7. Select **Debug → Run** from the Code window main menu. The image is executed and a prompt is displayed in the Output pane at the bottom of the Code window.
8. In the **StdIO** tab of the Output pane, enter the value 50 for the number of runs through the benchmark you want RealView Debugger to perform.
9. The results of the trace capture are displayed in the Analysis window.

Chapter 3

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 3-2
- *Using RealView Debugger RTOS extensions* on page 3-7
- *Connecting to the target and loading an image* on page 3-22
- *Associating threads with views* on page 3-27
- *Working with OS-aware images in the Process Control pane* on page 3-34
- *Using the Resource Viewer window* on page 3-41
- *Debugging your RTOS application* on page 3-47
- *Using CLI commands* on page 3-58.

3.1 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.

3.1.1 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.

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 3-34
- *Using the Resource Viewer window* on page 3-41
- *Using CLI commands* on page 3-58.

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 3-1 on page 3-5.

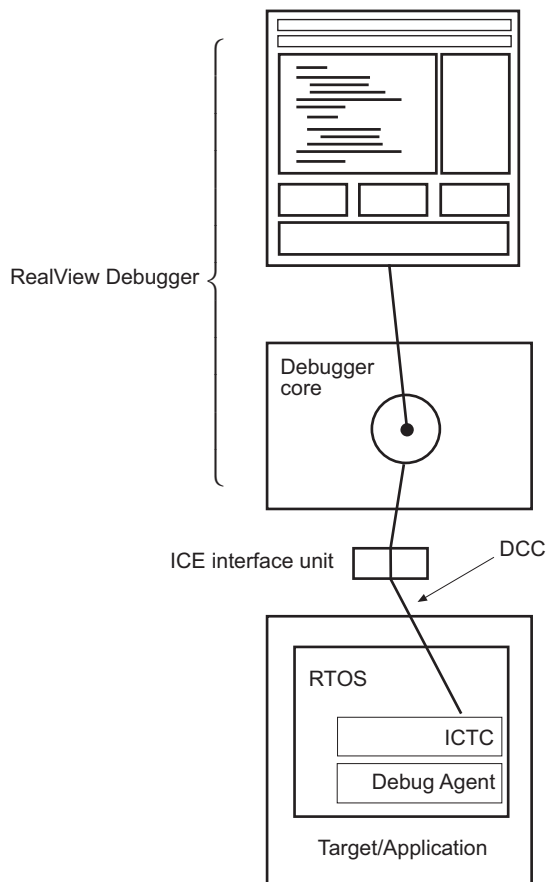


Figure 3-1 RealView Debugger and RTOS components

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.

3.2 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 3-8*
- *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file on page 3-11*
- *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file on page 3-13*
- *Managing configuration settings on page 3-20.*

3.2.1 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.

3.2.2 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 RealView ICE 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 RealView ICE 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 a RealView ICE interface unit in the chapter on configuring custom connections in *RealView Developer Kit v2.2 Debugger User Guide*.

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.



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.

———— Note ————

If the RealView ICE target vehicle is not visible in the Connection Properties window, you must add it before continuing with this section. See the chapter describing configuring a RealView ICE connection in *RealView ICE User Guide* for full details on how to do this.

3. Right-click on the connection that you want to use. This example uses RealView-ICE to access the ARM® JTAG debug tool.
4. Select **Connection Properties...** from the context menu to display the Connection Properties window, shown in Figure 3-2 on page 3-9.

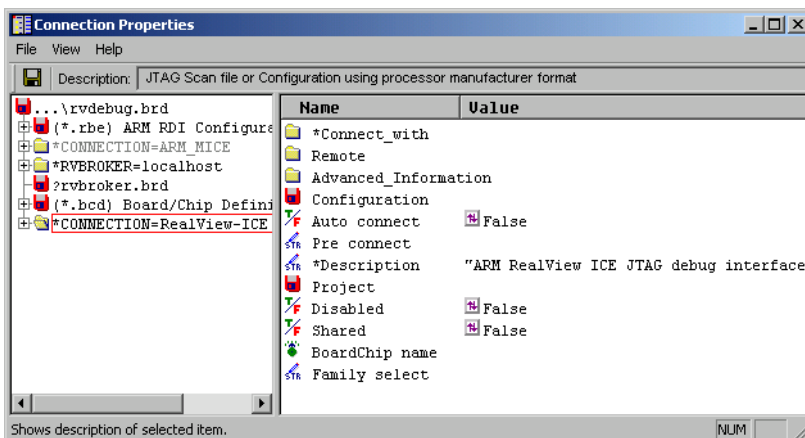


Figure 3-2 CONNECTION groups in the Connection Properties window

5. Right-click on the CONNECTION=RealView-ICE 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 3-3.

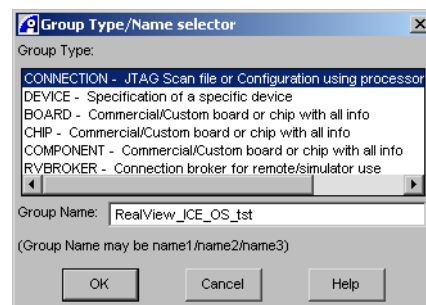


Figure 3-3 Specifying a new CONNECTION group

Leave the type of the new entry unchanged as CONNECTION.

In the Group Name data field change the name from RealView-ICE to something suitable for your target, for example RealView-ICE_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 RealView-ICE 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-NW**.
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 RealView ICE target is now displayed in the Connection Control window, shown in Figure 3-4.

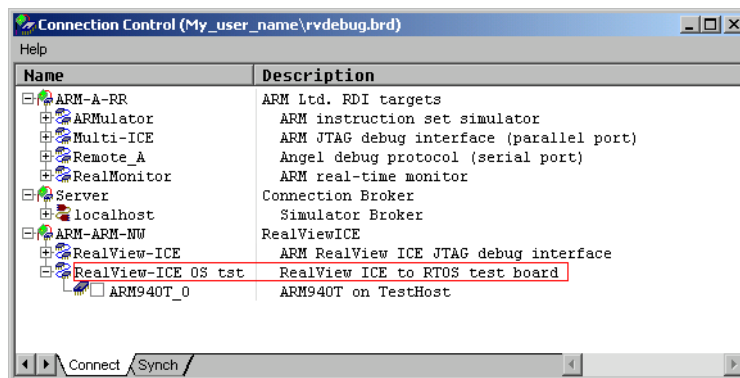


Figure 3-4 New connection in the Connection Control window

Configuring a RealView ICE interface unit

Ensure that the RealView ICE unit is configured as described in the chapter on configuring custom connections in *RealView Developer Kit v2.2 Debugger User Guide* before you continue with this section.

Remember the following when specifying settings for your hardware:

- Autoconfiguring the RealView ICE 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 RealView ICE 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 *RealView Developer Kit v2.2 Debugger User Guide*.

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 3-13.

3.2.3 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 3-4 on page 3-10.
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 3-7).
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 3-5.

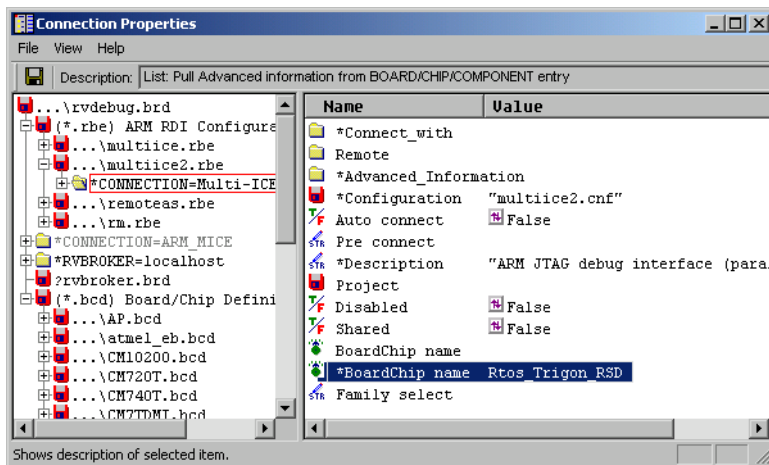


Figure 3-5 Referencing the RTOS .bcd file

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 3-19 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 3-22.

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 3-20 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 the chapter describing configuring custom targets in *RealView Developer Kit v2.2 Debugger User Guide*.

3.2.4 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 3-4 on page 3-10.
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 3-6 on page 3-14.

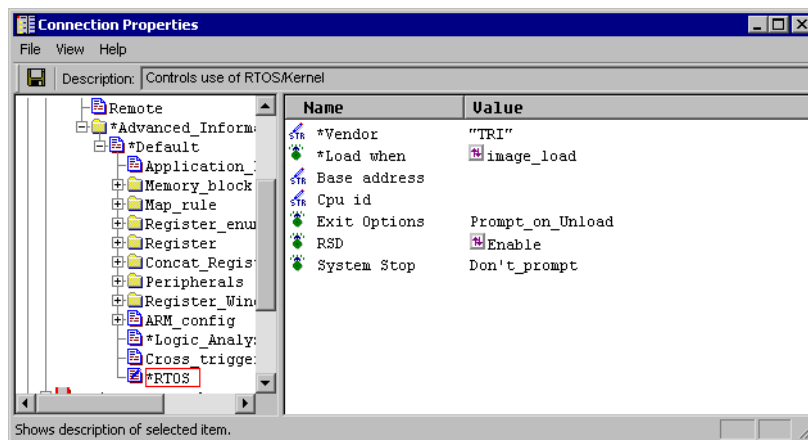


Figure 3-6 RTOS group in the Connection Properties window

As shown in Figure 3-6, 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 3-41 and *Working with OS-aware images in the Process Control pane* on page 3-34 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 3-34 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 3-17 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 3-7.

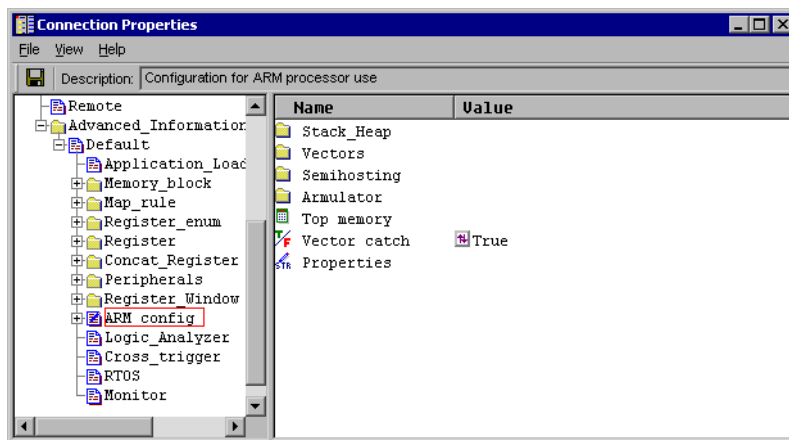


Figure 3-7 ARM_config group in the Connection Properties window

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 3-11.

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 3-11 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 3-4 on page 3-10.

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 3-8.

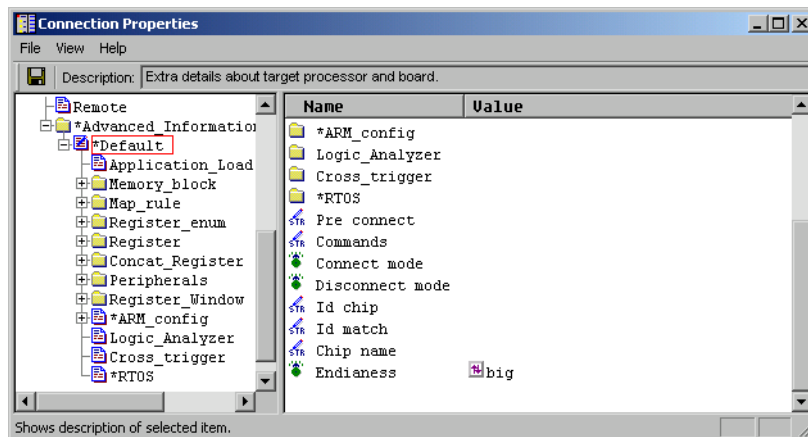


Figure 3-8 Default group in the Connection Properties window

As shown in Figure 3-8, 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 RealView ICE, 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 *RealView Developer Kit v2.2 Debugger User Guide*.

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 3-4 on page 3-10.
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 3-22.

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 3-20 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 the chapter describing configuring custom targets in *RealView Developer Kit v2.2 Debugger User Guide*.
- 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 the chapter describing configuring custom targets in *RealView Developer Kit v2.2 Debugger User Guide*.

3.2.5 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=RealView_ICE_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 the chapter that describes configuring custom targets in *RealView Developer Kit v2.2 Debugger User Guide*.

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=RealView_ICE_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.

3.3 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 3-23
- *RTOS Exit Options* on page 3-24
- *Interrupts when loading an image* on page 3-25
- *Resetting OS state* on page 3-25
- *Loading from the command line* on page 3-26.

3.3.1 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 3-7.
- create a new RTOS connection as described in *Creating a new RTOS-enabled connection* on page 3-8.
- configure your RTOS-enabled connection as described in either:
 - *Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file* on page 3-11
 - *Configuring an RTOS-enabled connection without a vendor-supplied .bcd file* on page 3-13.

3.3.2 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.

3.3.3 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 3-17.

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 3-24 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 the chapter describing connecting to targets in *RealView Developer Kit v2.2 Debugger User Guide* for full details on connect, and disconnect, modes.

3.3.4 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 3-17 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 3-9 on page 3-25.

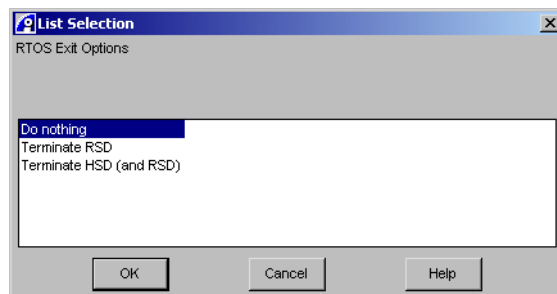


Figure 3-9 RTOS Exit Options selection box

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**.

3.3.5 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.

3.3.6 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.

3.3.7 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 (see the *RealView Developer Kit v2.2 Debugger User Guide* for details).

2. Provide the executable filename and any required arguments on the command line.

3.4 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*
- *Using the Cycle Threads button* on page 3-28
- *Working with the thread list* on page 3-30.

You can also work with threads in the Process Control pane, see *Working with OS-aware images in the Process Control pane* on page 3-34 for details.

3.4.1 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.

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. 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 3-30 for details.

3.4.2 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 Guide* for a full description of the **THREAD** command.

3.4.3 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* 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 3-30 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 3-38 for details.

3.4.4 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 3-10.

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				
0x00016e50	System Timer Thread 0		SUSP	0x00000000
0x00015508	ICTM	31	READY	0x00000000
0x000159a4	Debug Agent	3	READY	0x00000000
0x00013a34	thread_0	1	SLEEP	0x00000000
0x00013ac8	thread_1	16	SLEEP	0x00000000
0x00013b5c	thread_2	16	SUS_QUEUE	0x00000000
0x00013bf0	thread_3	8	SLEEP	0x00000000
0x00013c84	thread_4	8	SUS_SEM	0x00000000
0x00013d18	thread_5	4	SUS_EV	0x00000000
0x00013dac	thread_6	8	SUS_MX	0x00000000
*0x00013e40	thread_7	8	READY	0x00000000

Figure 3-10 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 3-32 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 3-10, 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 3-33 for details).

Current thread

As described in *The current thread* on page 3-27, RealView Debugger designates a thread to be the current thread when you are in RSD. In Figure 3-10 on page 3-30, 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 3-11 on page 3-32, 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 3-34 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 3-10 on page 3-30 shows two grayed threads (see *Special threads in the Thread tab* on page 3-40):

- 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 3-10 on page 3-30. 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 3-11.

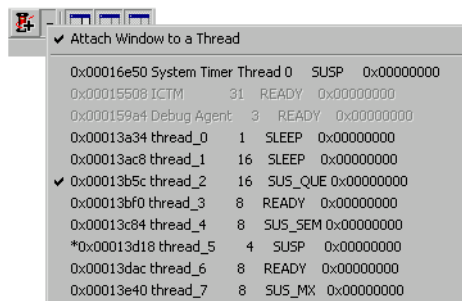


Figure 3-11 Example thread list in an attached window

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 3-38 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 3-12.

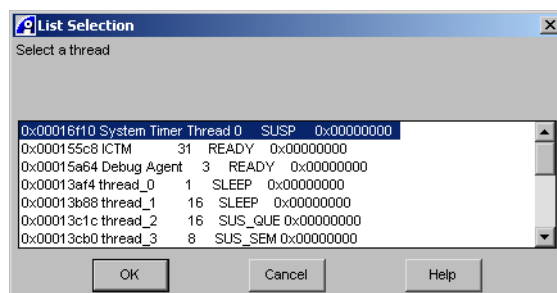


Figure 3-12 Thread selection box

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.

3.5 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 the chapter describing memory mapping in the <i>RealView Developer Kit v2.2 Debugger User Guide</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 3-38.

3.5.1 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 3-35
- *OS marker with an OS-aware image loaded* on page 3-35
- *OS marker for a running target with RSD fully operational* on page 3-36
- *OS marker status* on page 3-36
- *Context menu* on page 3-37.

OS marker initial state

If HSD or RSD is enabled, the **Process** tab contains the OS marker, shown in Figure 3-13. This is the initial state with no image loaded.

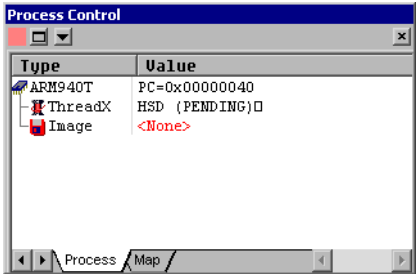


Figure 3-13 OS marker initial state in the Process Control pane (HSD)

In this state, the **Thread** tab and the **Cycle Threads** button are not available.

OS marker with 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 3-14.

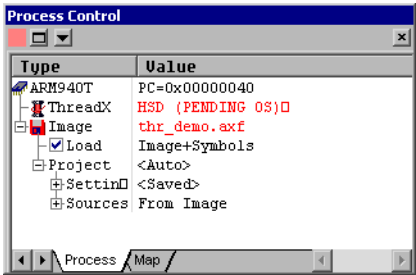


Figure 3-14 OS marker with loaded image in the Process Control pane (HSD)

Figure 3-14 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, the **Thread** tab and the **Cycle Threads** button are not available.

OS marker for a running target with RSD fully operational

Figure 3-15 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.

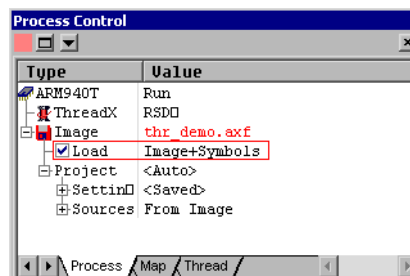


Figure 3-15 OS marker in the Process Control pane (RSD)

In this state, the **Thread** tab is available and the **Cycle Threads** button is enabled on the Connect toolbar.

OS marker status

Table 3-1 describes OS marker status in the Process Control pane.

Table 3-1 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 3-1 OS marker status in the Process Control pane (continued)

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 3-14 on page 3-35, 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 3-42.

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 3-13 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 3-48.

3.5.2 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 3-16.

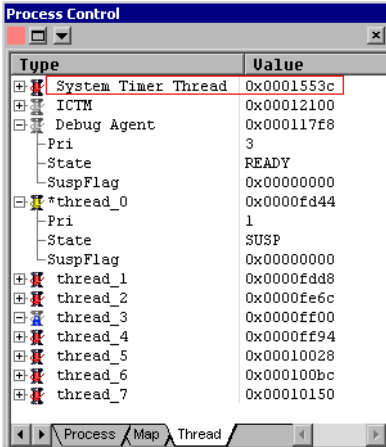






Figure 3-16 Thread tab in the Process Control pane

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 3-16 on page 3-38 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 3-27).

Attach Window to

Attach the Code window to this thread (see *Attaching windows to threads* on page 3-32).

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 3-4 for details.

3.6 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 3-42
- *Resource Viewer window interface components* on page 3-42
- *Using Action context menus* on page 3-45
- *Interaction of RTOS resources and RealView Debugger* on page 3-46.

3.6.1 Displaying the Resource Viewer window

To display this window, shown in Figure 3-17, select **View → Resource Viewer Window** from the Code window main menu.

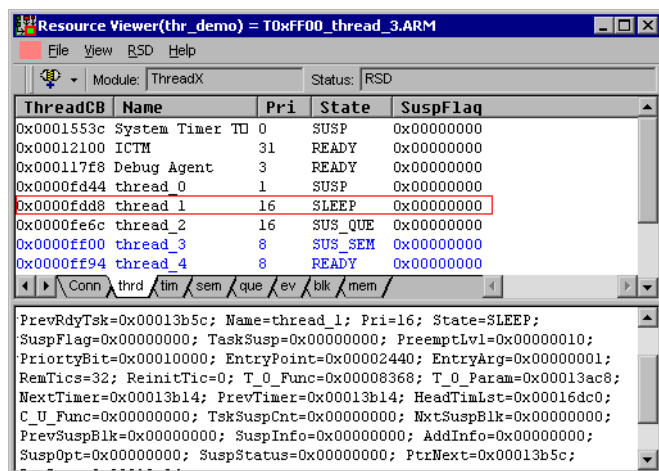


Figure 3-17 Resource Viewer showing thread details

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.

3.6.2 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.

3.6.3 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 3-13 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 3-34 for details.

Resources toolbar

This toolbar contains three controls:

Cycle Connections button

Not used

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 3-34 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 3-17 on page 3-41). 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 3-10 on page 3-30 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 3-17 on page 3-41).

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 3-17 on page 3-41).

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.

3.6.4 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 3-17 on page 3-41, 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 3-46 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 3-18 on page 3-46.

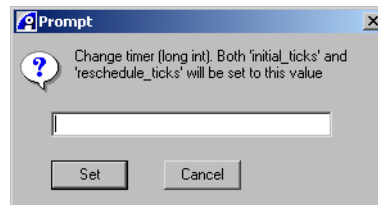


Figure 3-18 Action argument prompt

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 3-31 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 Guide* 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.

3.6.5 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.

3.7 Debugging your RTOS application

For full details on how to debug your applications using RealView Debugger, see *RealView Developer Kit v2.2 Debugger User Guide*. This section describes features specific to debugging multithreaded images in RealView Debugger. It contains the following sections:

- *About breakpoints*
- *Setting breakpoints* on page 3-49
- *Using the Set Address/Data Breakpoint dialog box* on page 3-51
- *Using the Break/Tracepoints pane* on page 3-53
- *Stepping threads* on page 3-54
- *Manipulating registers and variables* on page 3-56
- *Updating your debug view* on page 3-57.

3.7.1 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 3-34 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:

- the chapter describing working with breakpoints in *RealView Developer Kit v2.2 Debugger User Guide* for a full description of HSD breakpoints in RealView Debugger.
- the chapter describing memory mapping in *RealView Developer Kit v2.2 Debugger User Guide* for a full description of memory types and access rules.

3.7.2 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 3-58 for details).

3.7.3 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 3-19 on page 3-52.

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.

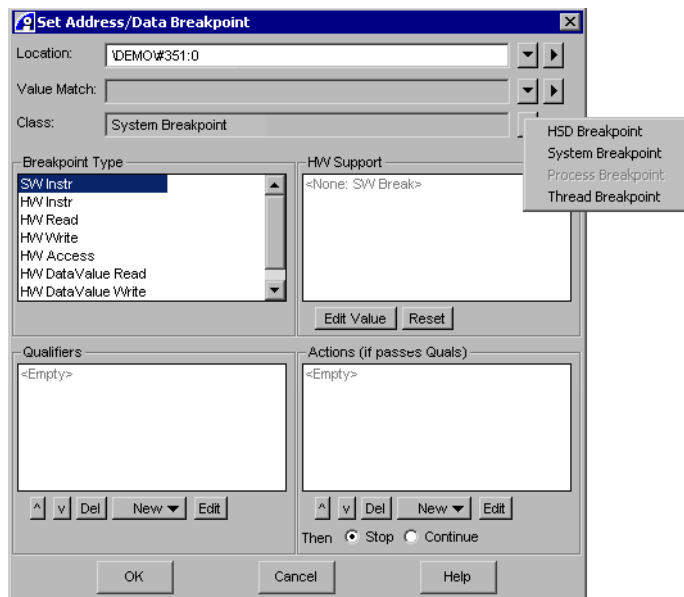


Figure 3-19 RTOS Breakpoint Class selector

————— **Note** —————

See the chapter describing working with breakpoints in *RealView Developer Kit v2.2 Debugger User Guide* for a full description of all the controls in this dialog box.

3. 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
- whether you are using tracepoints.

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.

3.7.4 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 3-20.

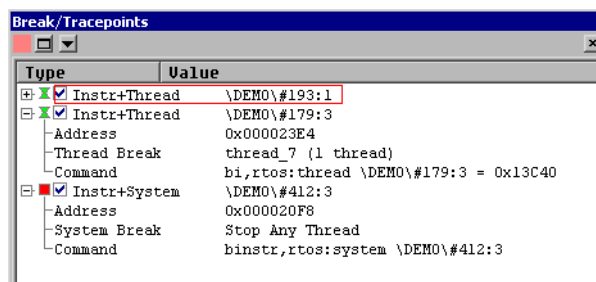


Figure 3-20 RTOS (RSD) breakpoints in the Break/Tracepoints pane

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 3-49 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 3-20 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 3-21 on page 3-54 shows three breakpoints are set in HSD mode. Here, the extra breakpoint details (available in RSD) are not included in the Break/Tracepoints pane.

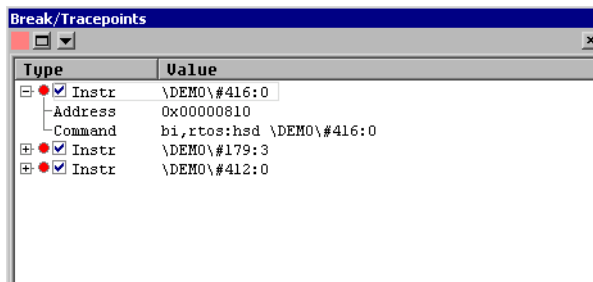
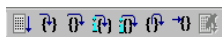


Figure 3-21 RTOS (HSD) breakpoints in the Break/Tracepoints pane

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 3-20 on page 3-53 and Figure 3-21 show that the Code window is unattached. In this case, the Code window displays the current thread.

See the chapter describing working with breakpoints in *RealView Developer Kit v2.2 Debugger User Guide* for a full description of the Break/Tracepoints pane.

3.7.5 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 3-22 represents three threads at different stages of execution.

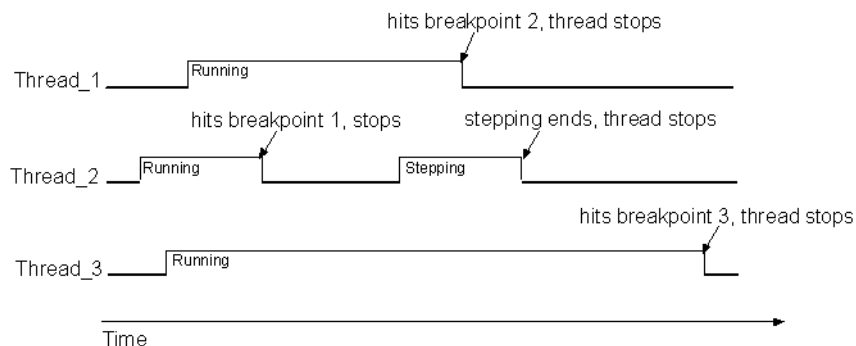


Figure 3-22 Stepping and stopping threads

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.

3.7.6 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 the chapter describing working with debug views in *RealView Developer Kit v2.2 Debugger User Guide* 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.
-

3.7.7 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 the chapter describing working with debug views in *RealView Developer Kit v2.2 Debugger User Guide* for details on working with the Memory and Watch panes, and a full description of all the options available from the **Pane** menus.

3.8 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 Guide*.

Appendix A

Setting up the Trace Hardware

This appendix describes how to set up the hardware for the trace configurations supported by RealView® Debugger. See *Getting started* on page 2-8 for details on how trace hardware components interact with RealView Debugger to enable you to perform tracing.

When setting up the trace capture system, do not exceed the timing specifications of the target *Embedded Trace Macrocell*™ (ETM) signals or of the trace capture hardware.

This appendix contains a section on *ARM RealView Trace and RealView ICE* on page A-2.

A.1 ARM RealView Trace and RealView ICE

The ARM RealView Trace analyzer is available from ARM Limited. When used with RealView ICE, it forms a complete trace solution. You connect it to the target board using the supplied ribbon cable and adaptor, and to the host workstation using a 10BaseT ethernet cable.

To set up your hardware to enable tracing in RealView Debugger as shown in Figure A-1, connect and configure the RealView ICE and RealView Trace units as described in the *RealView ICE User Guide*. For information on connecting to and configuring targets for use with RealView Debugger, see the chapter *Connection and Target Configuration* in the *RealView Developer Kit v2.2 Debugger User Guide*.

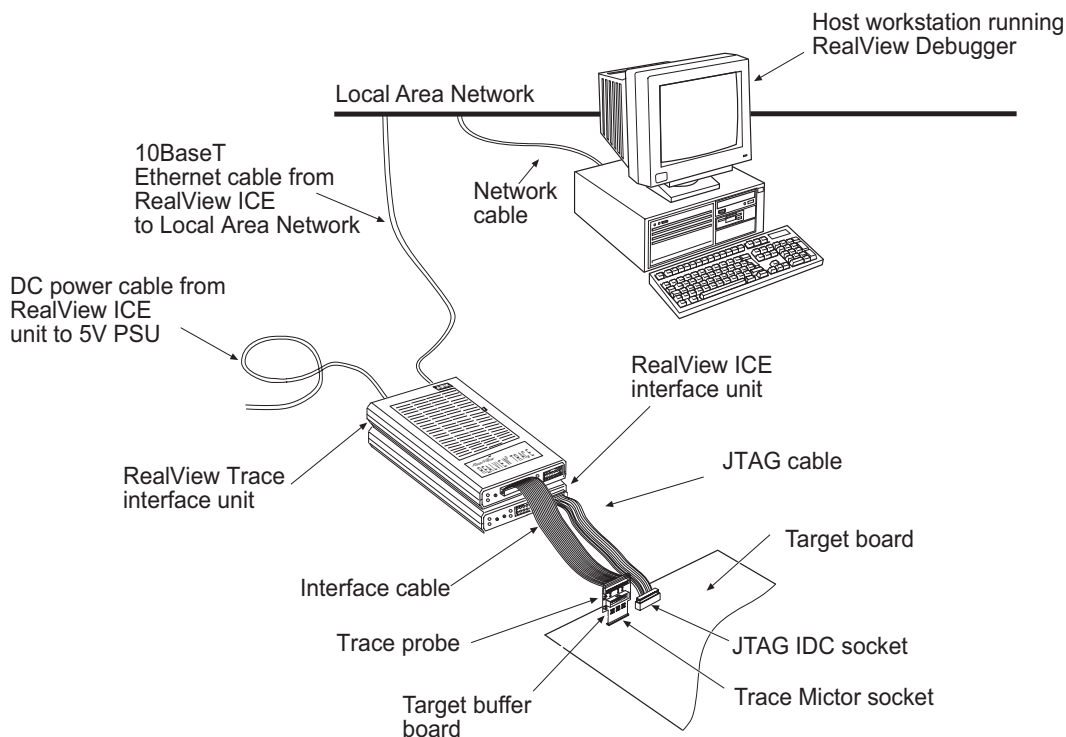


Figure A-1 Connections for RealView ICE and RealView Trace

Note

If the target buffer board contains a 20-way JTAG IDC connector, it is suggested that you use it. However, if the target board has only a Mictor socket, and no JTAG IDC socket, then you must use the JTAG IDC socket on the target buffer board.

Appendix B

Setting up the Trace Software

This appendix describes how to set up the software for the configurations of trace that are supported by RealView® Debugger. These instructions assume you have set up the hardware as described in Appendix A *Setting up the Trace Hardware*.

This appendix contains a section on *ARM RealView Trace and RealView ICE* on page B-2.

B.1 ARM RealView Trace and RealView ICE

This section describes how to set up RealView Debugger to connect to a combination of ARM RealView Trace and ARM RealView ICE. See the *RealView ICE User Guide* for more detailed information.

Note

RealView Trace support is available only on Windows platforms.

The section includes:

- *Installing and configuring RealView Trace and RealView ICE*
- *Adding a RealView ICE target vehicle and access-provider node* on page B-6.

B.1.1 Installing and configuring RealView Trace and RealView ICE

To install and configure RealView ICE and RealView Trace:

1. Install RealView ICE and connect up the hardware as described in the chapter *Using RealView Trace* in the *RealView ICE User Guide*.

Note

The RealView Trace software is included with RealView Debugger, and does not require a separate installation.

2. Start RealView Debugger.
3. Select **Target** → **Connection Properties** from the Code window main menu. The Connection Properties window appears.
4. If the RealView ICE target is present, continue at step 5.
If the RealView ICE target vehicle and access-provider node are not present, you must add them before you can continue with this procedure. See *Adding a RealView ICE target vehicle and access-provider node* on page B-6 for instructions on how to do this.
5. Right-click on the RealView-ICE access-provider node to display a context menu.
6. Select **Configure Device Info...** from the context menu:
 - If a configuration file already exists for RealView ICE, the RVConfig dialog box appears. Continue at step 8.
 - If no configuration file exists for RealView ICE, RealView Debugger displays the prompt dialog box that is shown in Figure B-1 on page B-3.

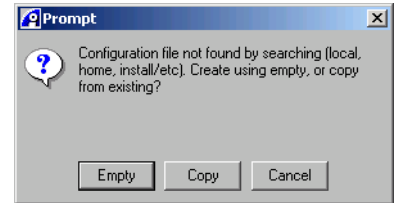


Figure B-1 Error when the configuration file is not found

7. Click **Empty** to create an empty configuration file. The Select Name of new file dialog box appears, as shown in Figure B-2.

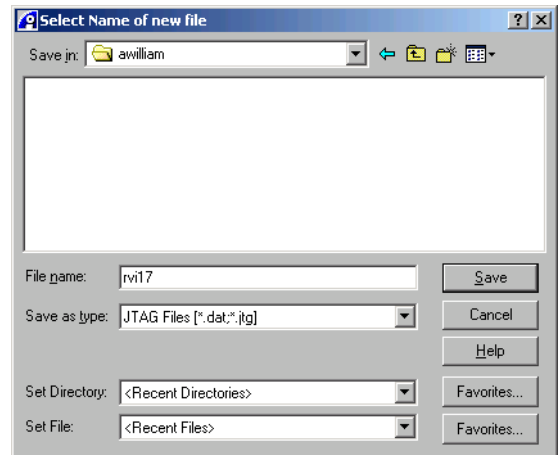


Figure B-2 Selecting where to store the new configuration file

To save the configuration file in your RealView Debugger home directory:

- a. Locate your RealView Debugger home directory.
 - b. Specify a filename to identify your RealView ICE interface unit, for example, `rvi.rvc`.
 - c. Click **Save**. The RVConfig dialog box appears.
8. Configure the RealView ICE target using the RVConfig dialog box shown in Figure B-3 on page B-4.

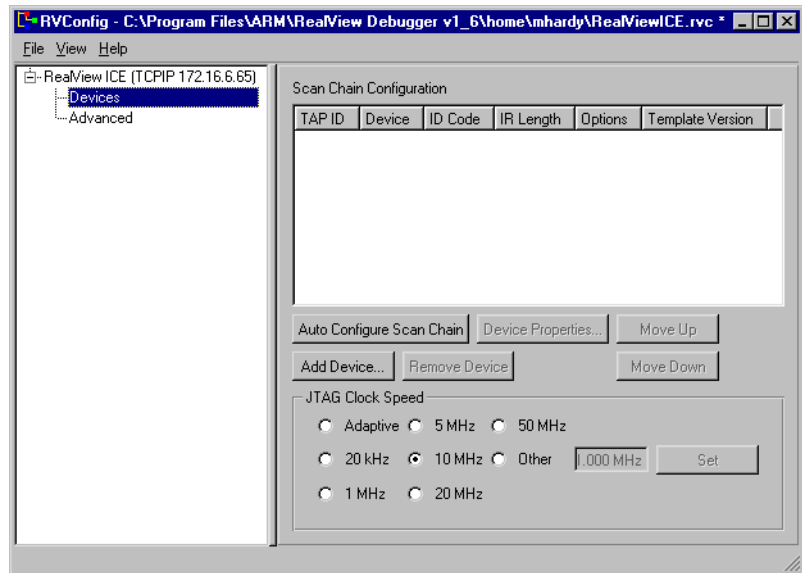


Figure B-3 The RVConfig dialog box

9. Select the **RealView ICE** node in the left-hand pane.
If your RealView ICE unit is connected to the USB port of your workstation, select **USB**. Continue at step 10.
If you want to use a RealView ICE unit that is available on the network:
 - a. Select **TCP/IP**.
 - b. Click the **Browse...** button.
 - c. Select the RealView ICE unit from the list.
 - d. Click **OK**.
10. Click the **Connect...** button. A **Devices** node is added to the tree diagram.
11. Select the **Devices** node and select **Auto Configure Scan Chain**. Each detected device is added to the **Scan Chain Configuration** list in the control pane, and is also added to the tree diagram. For detailed information on configuring a scan chain, see the chapter *Configuring a RealView ICE Connection* in the *RealView ICE User Guide*.
12. Set the JTAG clock speed by selecting the clocking that you want in the area at the bottom of the RVConfig dialog box. If the speed that you want to use is not available as a preset:
 - a. Select the **Other** button.

- b. Type the required speed. You can use a suffix of k or kHz to set a speed in kHz, or a suffix of M or MHz to set a speed in MHz. For example:
 - to set a clock speed of 20kHz, you can type 20kHz, or 20k, or 20000Hz, or 20000
 - to set a clock speed of 1MHz, you can type 1MHz, or 1M, or 1000kHz, or 1000k, or 1000000Hz, or 1000000.
 - c. Click on **Set**.
 - d. Select **File** → **Save** to save the configuration.
 - e. Select **File** → **Exit** to close the RVConfig dialog box.
13. Set the **Vendor** property of the RealView ICE connection to ARM:
 - a. Open the Connection Properties window by selecting **Target** → **Connection Properties** in RealView Debugger to open the Connection Properties window.
 - b. Expand the CONNECTION=RealView_ICE group.
 - c. Expand the Advanced_Information group.
 - d. Expand the Default group.
 - e. Click on the Logic_Analyzer item. The Logic_Analyzer settings are displayed in the right-hand pane, as shown in Figure B-4.

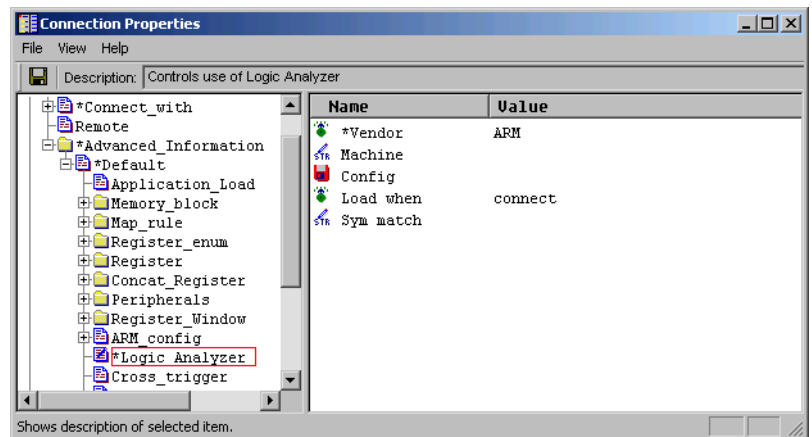


Figure B-4 Connection Properties window show Logic_Analyzer settings

- f. In the right-hand pane, right-click on **Vendor**, and select **ARM** from the context menu.
 - g. Select **File** → **Save and Close** to save the changes and close the Connection Properties window.

B.1.2 Adding a RealView ICE target vehicle and access-provider node

To add a RealView ICE target vehicle and access-provider node:

1. Select **Target** → **Connection Properties** from the Code window main menu. The Connection Properties window is displayed.
2. Right-click on the ...\\rvdebug.brd node at the root of the tree, that represents the board file, to display a context menu.
3. Select **Make New Group** from the context menu, as shown in Figure B-5.

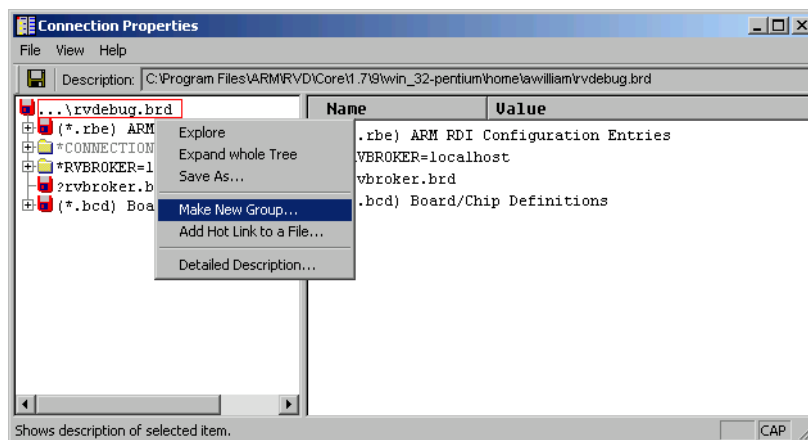


Figure B-5 Making a new group

The Group Type/Name selector window appears.

4. In the Group Type/Name selector window:
 - a. Choose **CONNECTION**
 - b. Set the **Group name** to the name that you want to use for the access provider node, such as RealView_ICE.
 - c. Click **OK**.

A new CONNECTION= node appears in the Connection Properties window, for example CONNECTION=RealView_ICE.

5. Expand the new CONNECTION= node, and select Connect_with.
6. In the right-hand pane, select the Manufacturer setting.
7. Right-click on the Manufacturer setting, and choose **ARM-ARM-NW - RealViewICE** from the context menu that appears, as shown in Figure B-6 on page B-7.

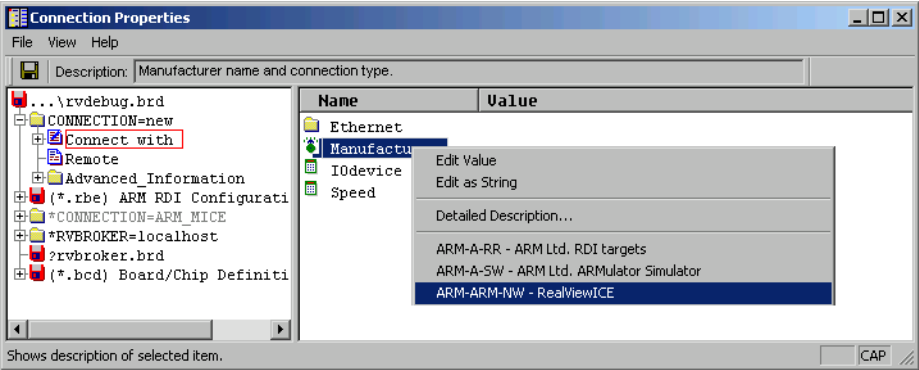


Figure B-6 Setting how to connect

- 8. Choose **File** → **Save and Close** to save the new configuration, and close the Connection Properties window.

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.

ADS

See ARM Developer Suite.

Agilent emulation module

A JTAG interface unit supported by RealView Debugger.

Agilent emulation probe

A JTAG interface unit supported by RealView Debugger.

Agilent logic analyzer

A trace capture hardware device supported by RealView Debugger.

Agilent traceport analyzer

A trace capture hardware device supported by RealView Debugger.

Angel

Angel is a software debug monitor that runs on the target and enables you to debug applications running on ARM-based hardware. Angel is commonly used where a JTAG emulator, such as Multi-ICE, is not available.

AAPCS *See* ARM Architecture Procedure Call Standard.

ARM Architecture Procedure Call Standard (AAPCS)

The ARM-Architecture Procedure Call Standard specifies a family of *Procedure Call Standard* (PCS) variants, to define how separately compiled and assembled routines can work together. The standard provides equal support for both ARM-state and Thumb-state to enable interworking. It favors small code-size, and provides functionality appropriate to embedded applications and high performance.

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 MultiTrace External collection unit for ARM Real-Time Trace.

ARM state 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.

See also Thumb state.

Asynchronous execution

Asynchronous execution 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.

Backtracing *See* Call Stack.

Big-endian Memory organization where the least significant byte of a word is at the highest address and the most significant byte is at the lowest address in the word.

See also Little-endian.

Board RealView Debugger uses the term *board* to refer to a target processor, memory, peripherals, and debugger connection method.

Board file The *board file* is the top-level configuration file, normally called `rvdebug.brd`, that references one or more other files.

Breakpoint A user defined point at which execution stops in order that a debugger can examine the state of memory and registers.

See also Hardware breakpoint and Software breakpoint.

Call Stack	This is a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.
Captive thread	<p>Captive threads are all threads that can be brought under debugger control. Special threads, called non-captive threads, are essential to the operation of <i>Running System Debug</i> (RSD) and so are not under the control of RealView Debugger. Non-captive threads are grayed out in the GUI.</p> <p><i>See also</i> Running System Debug.</p>
Conditional breakpoint	A breakpoint that halts execution when a particular condition becomes True. The condition normally references the values of program variables that are in scope at the breakpoint location.
Conditional tracepoint	<p>A type of tracepoint that enables you to set AND or OR conditions, counter conditions, and complex comparisons. These conditions can involve any supportable combination of trigger points, and start and end points and ranges.</p> <p><i>See also</i> Tracepoints.</p>
Context menu	<i>See</i> Pop-up menu.
Core module	<p>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.</p> <p><i>See also</i> Integrator</p>
Current Program Status Register (CPSR)	<i>See</i> Program Status Register.
DCC	<i>See</i> Debug Communications Channel.
Debug Agent (DA)	<p>The Debug Agent resides on the target to provide target-side support for <i>Running System Debug</i> (RSD). The Debug Agent can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the <i>debug communications channel</i> (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state.</p> <p><i>See also</i> Running System Debug.</p>

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).

Deprecated

A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features are to be removed in future versions of the product.

Digital Signal Processor (DSP)

DSPs are special processors designed to execute repetitive, maths-intensive algorithms. Embedded applications might use both ARM processor cores and DSPs.

Doubleword

A 64-bit unit of information.

DSP

See Digital Signal Processor.

DWARF

See Debug With Arbitrary Record Format.

ELF

Executable and Linking Format. ARM code generation tools produce objects and executable images in ELF format.

Embedded Trace Buffer (ETB)

The Embedded Trace Buffer provides logic inside the core that extends the information capture functionality of the Embedded Trace Macrocell.

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*.

ETB

See Embedded Trace Buffer.

ETM See Embedded Trace Macrocell.

ETV See Extended Target Visibility.

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.

FIFO First-In-First-Out.

Filtering A facility that enables you to refine the results of a trace capture that has already been performed. This is useful if you want to refine your area of interest within the display.

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.

General Purpose Input/Output (GPIO)

This refers to the pins on an ASIC that are used for I/O. Some of these GPIO pins can be multiplexed to extend the trace port width.

GPIO See General Purpose Input/Output.

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 and Software breakpoint.

HSD See Halted System Debug.

IEEE Std. 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. <i>Core modules</i> are available that contain the processor and local memory.
Joint Test Action Group (JTAG)	An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a <i>Joint Test Action Group</i> (JTAG) interface port to communicate with processors. For more information see IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).
JTAG	<i>See</i> Joint Test Action Group.
JTAG interface unit	A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the EmbeddedICE logic and the ETM.
Little-endian	Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word. <i>See also</i> Big-endian.
Multi-ICE	A JTAG-based tool for debugging embedded systems.
Pop-up menu	Also known as <i>Context menu</i> . A menu that is displayed temporarily, offering 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)	<i>Program Status Register</i> (PSR), containing some 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.
RDI	<i>See</i> Remote Debug Interface.

RealView ARMulator ISS (RVISS)

The most recent version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite. It communicates with a debug target using RV-msg, through the RealView Connection Broker interface, and RDI.

See also RDI and RealView Connection Broker.

RealView Compilation Tools (RVCT)

RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of *RISC* processors.

RealView Connection Broker

RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system. It also enables you to make multiple connections to the simulator.

See also RealView ARMulator ISS.

RealView Debugger Trace

Part of the RealView Debugger product that extends the debugging capability with the addition of real-time program and data tracing. It is available from the Code window.

RealView ICE (RVI)

A JTAG-based debug solution to debug software running on ARM processors.

RealView Trace (RVT)

Works in conjunction with ARM RealView ICE to provide real-time trace functionality for software running in leading edge System-on-Chip devices with deeply embedded processor cores.

Remote_A

Remote_A is a software protocol converter and configuration interface. It converts between the RDI 1.5 software interface of a debugger and the Angel Debug Protocol used by Angel targets. It can communicate over a serial or Ethernet interface.

Remote Debug Interface (RDI)

The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a simulator running on the host (for example, RVISS)
- a debug monitor running on hardware that is based on an ARM core accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, RealView ICE or Multi-ICE).

RSD

See Running System Debug.

RTOS

Real Time Operating System.

Running System Debug (RSD)

Used for RTOS aware debugging, *Running System Debug* (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a *Debug Agent* (DA) that resides on the target. The Debug Agent is scheduled along with other tasks in the system.

See also Debug Agent and Halted System Debug.

RVCT

See RealView Compilation Tools.

RVISS

See RealView ARMulator ISS.

Scan chain

A scan chain is made up of serially-connected devices that implement boundary-scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain. Processors might contain several shift registers to enable you to access selected parts of the device.

Scope

The range within which it is valid to access such items as a variable or a function.

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 and Hardware breakpoint.

Software Interrupt (SWI)

An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.

SPSR

Saved Program Status Register.

See also Program Status Register.

Status lines

Refers to those rows of trace output in the RealView Debugger Analysis window that are for status-only purposes, such as Trace Pause, and describe information about the processor cycle.

SWI

See Software Interrupt.

Synchronous execution

Synchronous execution 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

See 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.

See also Test Access Port and IEEE1149.1.

Target

The target board, 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, target knowledge, such as memory mapping, lists, rule processing, board-files and .bcd files, and data structures to track the target environment.

Tektronix Logic Analyzer (TLA)

A trace capture hardware device supported by RealView Debugger.

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.

See also ARM state.

TLA

See Tektronix Logic Analyzer.

TPA *See* Trace Port Analyzer.

TPM *See* Trace Port Multiplexer.

Trace capture hardware

An external device that stores the information from the trace port. Some processors contain their own on-chip trace buffer, where an external device is not required.

Trace Port Analyzer (TPA)

An external device that stores the information from the trace port. This information is compressed so that the analyzer does not have to capture data at the same bandwidth as that of an analyzer monitoring the core buses directly.

Trace Port Multiplexer (TPM)

A device that combines the output from two traceports into a single traceport output. This enables you to use a single Mictor connector for two traceports.

Tracepoint

A tracepoint can be a line of source code, a line of assembly code, or a memory address. In RealView Debugger, you can set a variety of tracepoints to determine exactly what program information is traced.

Tracepoint unit

A unit within a conditional tracepoint, similar to an unconditional tracepoint, which combines with other tracepoint units to create the conditional tracepoint.

Tracing

The real-time recording of processor activity (including instructions and data accesses) that occurs during program execution. Trace information can be stored either in a trace buffer of a processor, or in an external trace hardware unit. Captured trace information is returned to the Analysis window in RealView Debugger where it can be analyzed to help identify a defect in program code.

Trigger

In the context of breakpoints, a trigger is the action of noticing that the breakpoint has been reached by the target and that any associated conditions are met. In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.

TVS

See Target Vehicle Server.

Unconditional tracepoint

A type of tracepoint that enables you to set trigger points, trace start and end points, or trace ranges for memory and data accesses, that do not depend on conditions.

See also Tracepoints.

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 watches you have defined.
Watchpoint	In RealView Debugger, this is a hardware breakpoint.
Word	A 32-bit unit of information.

