RealView Debugger
Extensions User Guide

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

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 2002</td>
<td>A</td>
<td>RealView Debugger v1.5 release</td>
</tr>
<tr>
<td>September 2002</td>
<td>B</td>
<td>RealView Debugger v1.6 release</td>
</tr>
<tr>
<td>February 2003</td>
<td>C</td>
<td>RealView Debugger v1.6.1 release</td>
</tr>
<tr>
<td>September 2003</td>
<td>D</td>
<td>RealView Debugger v1.6.1 release for RVDS v2.0</td>
</tr>
<tr>
<td>January 2004</td>
<td>E</td>
<td>RealView Debugger v1.7 release for RVDS v2.1</td>
</tr>
</tbody>
</table>

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 Open Access. This document has no restriction on distribution.

Product Status

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

Web Address

http://www.arm.com
# Contents

**RealView Debugger Extensions User Guide**

## Preface

About this book ................................................................................................................ vi
Feedback .......................................................................................................................... xii

## Chapter 1  Introduction to RealView Debugger Extensions

1.1 About RealView Debugger extensions ................................................................. 1-2
1.2 Licensing ............................................................................................................. 1-6
1.3 Supported platforms ......................................................................................... 1-7
1.4 Supported hardware ......................................................................................... 1-8

## Chapter 2  Tracing with RealView Debugger

2.1 About tracing with RealView Debugger ............................................................ 2-2
2.2 Getting started .................................................................................................. 2-7
2.3 Configuring the ETM ....................................................................................... 2-11
2.4 Configuring trace capture ............................................................................... 2-21
2.5 Using the Analysis window ............................................................................. 2-44
2.6 Examples of using trace in RealView Debugger ............................................. 2-95

## Chapter 3  DSP Support

3.1 About DSPs and RealView Debugger DSP support ......................................... 3-2
3.2 Using the DSP .................................................................................................... 3-4
Chapter 4  RTOS Support
4.1 About Real Time Operating Systems ................................................. 4-2
4.2 Using RealView Debugger RTOS extensions ....................................... 4-7
4.3 Connecting to the target and loading an image ................................... 4-22
4.4 Associating threads with views ............................................................ 4-27
4.5 Working with threads in the Process Control pane .............................. 4-34
4.6 Using the Resource Viewer window .................................................... 4-40
4.7 Debugging your RTOS application ...................................................... 4-46
4.8 Using CLI commands ........................................................................ 4-58

Chapter 5  Working with Multiple Target Connections
5.1 Overview of multiple target connections in RealView Debugger .......... 5-2
5.2 The RealView Debugger multiprocessor architecture .......................... 5-3
5.3 Managing multiple targets ................................................................. 5-13
5.4 Display coherency .............................................................................. 5-37
5.5 Processor execution synchronization ................................................... 5-45

Appendix A  Setting up the Trace Hardware
A.1 ARM MultiTrace and ARM Multi-ICE .................................................... A-2
A.2 ARM RealView Trace and RealView ICE .............................................. A-4
A.3 Agilent 16600 or 16700 logic analyzer and Emulation Probe ............... A-5
A.4 Agilent 16600 or 16700 logic analyzer and Multi-ICE ......................... A-9
A.5 Agilent Emulation Probe and Trace Port Analyzer (E5904B) .............. A-12
A.6 Tektronix TLA 600 or TLA 700 logic analyzer and Multi-ICE ............... A-15

Appendix B  Setting up the Trace Software
B.1 ARM MultiTrace and ARM Multi-ICE .................................................... B-2
B.2 Embedded Trace Buffer and ARM Multi-ICE ...................................... B-6
B.3 ARM RealView Trace and RealView ICE ............................................ B-10
B.4 ARM Multi-ICE for XScale ................................................................. B-15
B.5 Agilent 16600 or 16700 Logic Analyzer and Emulation Probe ............ B-17
B.6 Agilent 16600 or 16700 Logic Analyzer and ARM Multi-ICE ............... B-24
B.7 Agilent Trace Port Analyzer and Agilent Emulation Probe .................. B-27
B.8 Tektronix TLA 600 or TLA700 and ARM Multi-ICE ............................ B-30
B.9 Simulators using the Simulator Broker connection ............................ B-33

Glossary
Preface

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

- About this book on page vi
- Feedback on page xii.
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
- the RealView Debugger Digital Signal Processor (DSP) extension, available by license only
- the RealView Debugger multiprocessor extension, available by license only.

This book only describes the debugger extensions. Refer to the other books in the RealView Debugger 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, and have some experience with tracing, debugging multiple processor targets, DSP or RTOS development, as appropriate.

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**

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

**RTOS support**

You should have some experience with debugging an RTOS application.

**DSP support**

You should have some experience with debugging programs that run on a DSP target.
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 4 RTOS Support**
Read this chapter for a description of the support RealView Debugger provides for debugging an RTOS application.

**Chapter 3 DSP Support**
Read this chapter for a description of the support RealView Debugger provides for debugging a program that runs on a DSP target.

**Chapter 5 Working with Multiple Target Connections**
Read this chapter for details on the RealView Debugger features that enable you to make more than one connection at a time. This is useful when you are debugging multitasking applications that are running on multiple processors.

**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.

**Glossary**
Refer to this for explanations of terms used in this book.
Typographical conventions

The following typographical conventions are used in this book:

italic 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.

monospace as monospace Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic 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.

Timing diagram conventions

This figure describes the conventions of the event timing diagrams in this manual:

TRUE to FALSE
FALSE to TRUE, uncertain timing
Momentary event
Stable state
Stable state to undefined
Stable state to another stable state
Part of sequence omitted for clarity
Arrow indicating related events

Key to event timing diagram conventions
Shaded areas represent periods when the value is undefined, for example because a state change requires changes to many data structures. Shaded areas are also used when the precise time the state changes, relative to other events shown, is variable.

Momentary events are used to represent triggers, for example a software interrupt.

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 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 Debugger documentation suite. Other books in this suite include:

- RealView Debugger v1.7 Essentials Guide (ARM DUI 0181)
- RealView Debugger v1.7 User Guide (ARM DUI 0153).
- RealView Debugger v1.7 Project Management User Guide (ARM DUI 0227)
- RealView Debugger v1.7 Target Configuration Guide (ARM DUI 0182)

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

For details on using RealView ARMulator® ISS, refer to the following documentation for more information:

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

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

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

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

- RealView ICE User Guide (ARM DUI 0155)
- ARM Agilent Debug Interface User Guide (ARM DUI 0158)
- Multi-ICE® Version User Guide (ARM DUI 0048)
- ARM MultiTrace™ User Guide (ARM DUI 0150).
Other publications

For a comprehensive introduction to ARM architecture see:


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


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


For more information about the JTAG standard, see:


For more information about Oak and TeakLite processors from the DSP Group see http://www.dspg.com.

Contact information for the MaxSim Simulator from AXYS Design Automation, Inc. is available at http://www.axysdesign.com.

Refer to the following publications for additional information about the Agilent analyzers described in this manual:


To access these documents, see the website http://www.agilent.com.

Refer to the following publication for additional information about the Tektronix Trace controller software described in this book:


Refer to the following websites for additional information about the Tektronix analyzers described in this book:

- the Tektronix web site, http://www.tek.com
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.
Preface
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 1-6
- *Supported platforms* on page 1-7
- *Supported hardware* on page 1-8.
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
- Chapter 4 DSP Support on page 1-4
- Chapter 5 Working with Multiple Target Connections on page 1-5.

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 Debugger v1.7 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 can perform tracing using trace hardware or a simulator. However, 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
- complex trace control points 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 enable you to:

- configure tracing options to apply to all trace captures you perform
- view tracing and profiling information, using up to six different views
- filter the results of a trace capture
- search for a specific item of trace information
- manipulate the display of trace information.
Note

It is recommended that you read Examples of using trace in RealView Debugger on page 2-95 before attempting to perform tracing on your own program. This section provides examples of using trace to solve typical development problems, and does not assume 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. You must obtain the RealView Debugger plugins for the RTOS you are using before you can use this extension.

Note

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. Select Goto RTOS Awareness Downloads from the Code window Help menu for more information.

RealView Debugger supports different debugging modes:

Halted System Debug

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

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

Running System Debug

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

RSD is only available where supported by your debug target. It relies on having RealView Debugger RTOS extensions installed and is not provided as part of the base product.

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

**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.

### 1.1.3 Chapter 4 DSP Support

This chapter describes the *Digital Signal Processor* (DSP) support available in RealView Debugger. It describes the type of support provided when using either a simulator or real DSP hardware.

**Installing DSP support**

Be aware of the following:

- Choose a Custom installation to install DSP support to ensure that the required JTAG files are available to enable connection using Multi-ICE® direct connect. You do not require a DSP license to connect to these targets.
  
  Currently, DSP support works only with remote targets containing an ARM7TDMI® core when connected through Multi-ICE direct connect.

- Choose a Custom installation to install DSP support if you want to use the DSP Group Oak DSP/TeakLite DSP, or the Motorola M56621 DSP. Do this to ensure that the required JTAG files are available. You must obtain a DSP license to connect to these targets.

- If you are using a DSP simulator, for example the MaxSim *System-on-Chip (SoC)* simulator from AXYS Design Automation, Inc., you must obtain a DSP license.
You can use RealView ICE to connect to a target that incorporates DSP hardware with a suitable JTAG configuration. However, it is not currently possible to connect to DSP cores using RealView ICE.

1.1.4 Chapter 5 Working with Multiple Target Connections

This chapter describes features within RealView Debugger that support multiprocessor debugging. It describes how RealView Debugger is used to debug mixed core systems and to synchronize processor operations. You must obtain the appropriate license before you can use this extension.
1.2 Licensing

To use the multiprocessor or DSP extension, you must have a valid license. You do not require a separate license to use Trace.

The RTOS extension is not licensed by ARM, but you must obtain enabling software for your chosen RTOS (see Chapter 4 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.
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
- Hardware for DSP 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 buffer capability so that you can view the contents of a trace buffer. For details on the processors supported by the tracing extension, see System requirements on page 2-3.

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.

1.4.3 Hardware for DSP support

The DSP-support extension of RealView Debugger is designed for use with only DSP Group processors:

- Oak
- TeakLite.

___ Note ___

You can use RealView ICE to connect to a target that incorporates DSP hardware with a suitable JTAG configuration. However, at this stage of development, it is not possible to connect to DSP cores using RealView ICE.
Chapter 2
Tracing with RealView Debugger

This chapter describes how to generate trace data using RealView® Debugger, and how to analyze the trace output using the Analysis window. It contains the following sections:

- About tracing with RealView Debugger on page 2-2
- Getting started on page 2-7
- Configuring the ETM on page 2-11
- Configuring trace capture on page 2-21
- Using the Analysis window on page 2-44
- Examples of using trace in RealView Debugger on page 2-95.
2.1 About tracing with RealView Debugger

This section describes the system requirements for performing trace with RealView Debugger, and the benefits and limitations of each method. It contains the following sections:

- Overview
- System requirements on page 2-3
- Available resources on page 2-4.

2.1.1 Overview

RealView Debugger enables you to perform tracing on your program. You can view a historical, non-intrusive trace of instructions and data accesses in real-time. This is useful when you are trying to identify a defect in your program code. Tracing is typically performed when a problem results from some interaction between application software and hardware, that occurs while your program 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.

RealView Debugger enables you to set conditions for generating trace information while your program is still running. For example, you can:

- Start and stop tracing on a tracepoint.
- Define a tracepoint as a range of addresses in which tracing of instructions and/or data will occur.
- Define a trigger. A trigger is an event that instructs the debugger to stop collecting trace as soon as the buffer is full 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.

Tracepoints can be:

**Simple**

These include individual trigger points, trace start and end points, and trace ranges for instruction and data accesses. See Setting simple tracepoints on page 2-22 for details.

**Complex**

These include AND or OR conditions, conditions on the number of executions, and complex comparisons. See Setting complex tracepoints on page 2-29 for details.

You can set the tracepoints from within the Code window, then view and analyze the results of the capture using the Analysis window. The Analysis window provides access to most of the tracing functionality, and enables you to view the captured trace information using any of six tabbed view types, including a **Profile** tab, which displays a statistical analysis of your trace information.
2.1.2 System requirements

RealView Debugger supports tracing with either trace hardware or a hardware simulator. This section describes the system requirements for both types of tracing. It contains the following sections:

- Trace hardware
- Simulators on page 2-4.

Trace hardware

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

- A trace solution. This can be either:
  - An ETM-enabled processor and a trace capture hardware device (See ETM trace solutions)
  - An on-chip trace buffer solution (see On-chip trace buffer solutions on page 2-4).
- A Joint Test Action Group (JTAG) interface unit, which can be one of:
  - ARM® RealView ICE version 1.1 and above
  - ARM Multi-ICE® version 2.0 and above
  - Agilent Emulation Module
  - Agilent Emulation Probe.

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 2-7.

ETM trace solutions

If you are using an Embedded Trace Macrocell™ (ETM) trace solution, you must have the following components:

- an ETM-enabled ARM processor
- a trace capture hardware device, which can be one of:
  - ARM RealView Trace unit (used for tracing in conjunction with ARM RealView ICE)
  - ARM MultiTrace™ unit
  - Agilent 16600 or 16700 logic analyzer
  - Agilent Trace Port Analyzer
— Tektronix TLA 600 or TLA 700 logic analyzer.

**On-chip trace buffer solutions**

RealView Debugger trace supports the following on-chip trace buffer solutions:

- ARM On-Chip Trace (ETM™)
- DSP-Group Oak
- DSP-Group TeakLite
- Motorola M56621
- Intel® XScale™.

**Simulators**

If you do not have trace hardware, you can use RealView ARMulator® ISS (RVISS) for basic instruction tracing only.

2.1.3 **Available resources**

This section describes the resources that are available for each tracing method. It contains the following sections:

- Trace hardware with an ETM-enabled ARM processor
- Trace hardware with non-ARM processors on page 2-6
- RealView ARMulator ISS on page 2-6.

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 Application-Specific Integrated Circuit (ASIC) designer. Three 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, and Large, and are described in Table 2-1 on page 2-5.
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.

The standard five JTAG interface pins are also required to set up the ETM.

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

--- **Note** ---
RealView Debugger does not detect whether all the optional features of the ETM, such as FIF0FULL, have been connected within the device containing the ETM. To determine this, refer to the documentation that accompanies your device.

---

### Table 2-1 ETM configurations

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pairs of address comparators</td>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Data comparators</td>
<td>0</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Memory map decoders</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Counters</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>State machine present</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>External inputs</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>External outputs</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>FIF0FULL present</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FIFO depth (bytes)</td>
<td>9 (ETM7™)</td>
<td>18 (ETM7)</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>10 (ETM9)</td>
<td>20 (ETM9)</td>
<td></td>
</tr>
<tr>
<td>Trace packet width</td>
<td>4/8</td>
<td>4/8/16</td>
<td>4/8/16</td>
</tr>
</tbody>
</table>
Trace hardware with non-ARM processors

If you are using a non-ARM processor, you can perform only basic instruction tracing. You can set trigger points, and start and end tracepoints (see Setting simple tracepoints on page 2-22 for details), but not trace ranges or complex tracepoints.

Note

When debugging XScale, you cannot set tracepoints.

RealView ARMulator ISS

If you are using RVISS, only basic instruction and data tracing support is available. You can set only trigger points, and start and end tracepoints (see Setting simple tracepoints on page 2-22 for details).
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 program after you have configured your system. It contains the following sections:

- Using the examples
- Overview of the trace hardware setup
- RealView Debugger tracing procedure on page 2-9.

2.2.1 Using the examples

As an introduction to the RealView Debugger tracing and profiling features, refer to Examples of using trace in RealView Debugger on page 2-95. These examples do not require any experience of using RealView Debugger, and they demonstrate the benefits of using trace to solve typical development problems. It is recommended that you perform these examples before using trace on your own program.

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 System requirements on page 2-3).
2. Ensure that RealView Debugger is properly connected to your target.
   - If you are using a simulator instead of trace hardware, ensure that your connection is correctly configured. See Simulators using the Simulator Broker connection on page B-33 in Appendix B Setting up the Trace Software for details.
   - If you are using trace hardware, you must first ensure that the components are connected and configured properly. 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 connect to and configure your chosen target.

2.2.2 Overview of the trace hardware setup

Figure 2-1 on page 2-8 shows an example of how RealView Debugger can be joined with ARM hardware to enable tracing capabilities.

---

**Note**

If you are using a non-ARM processor, refer to your processor documentation for details on how it can be integrated with other trace hardware.
The hardware elements that provide the capability to trace instructions and data accesses in real time are:

**ASIC that supports trace**

This can be one of:

**ARM CPU macrocell with an ETM (shown in Figure 2-1)**

This is an ARM-family CPU that contains EmbeddedICE® logic, and is ETM-enabled. 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 System requirements on page 2-3.

**ARM or DSP-group processor on-chip trace buffer**

This ASIC uses an on-chip trace buffer to store trace information. The processor stores control flow changes in the on-chip trace buffer and then extrapolates the trace data from this. For details on how this processor interacts with other trace hardware elements, see the documentation that accompanies the processor.
XScale  This is a processor core based on the Intel XScale Microarchitecture core. The XScale core can only trace instructions, and does not trace data.

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 System requirements on page 2-3.

JTAG interface
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 units that are supported by RealView Debugger, see System requirements on page 2-3.

2.2.3  RealView Debugger tracing procedure

It is recommended that you familiarize yourself with the tracing procedure before trying to perform trace capture on your own program using RealView Debugger. It is also suggested that you refer to Examples of using trace in RealView Debugger on page 2-95 to see how trace can be used in a typical development environment. The complete procedure for performing tracing on your program is as follows:

1.  Ensure that your hardware and software are properly connected and configured (see Appendix A Setting up the Trace Hardware and Appendix B Setting up the Trace Software).

2.  Start RealView Debugger and connect to a target that supports trace (see System requirements on page 2-3 for a list of supported targets).

3.  Load the image you want to analyze into RealView Debugger. For details on loading an image, see the chapter Working with Images in the RealView Debugger v1.7 User Guide.

4.  Change the general tracing configuration options in one, or both, of the following ways:
   a.  For ETM-enabled targets, use the Configure ETM dialog. See Configuring the ETM on page 2-11 for complete details on completing this dialog.
   b.  For non-ETM-enabled targets, the Configure ETM dialog is not available, and you must use the options in the Edit menu of the Analysis window. See Configuring trace options on page 2-48 for details.
5. Determine the area of interest in your source file for which you want to perform tracing, and set tracepoints accordingly. See Configuring trace capture on page 2-21.

6. Execute your program. The results of the trace capture are returned to the Analysis window, where you can analyze the results. See Using the Analysis window on page 2-44 for detailed information on using this window.

--- Note ---

The availability of tracing resources, and the options within the Analysis window, are dependent on your system configuration. See Available resources on page 2-4 for a description of which resources are available with each configuration.

This chapter describes how to perform trace-related operations using the RealView Debugger GUI. However, you can also perform many of the same operations using the RealView Debugger Command-Line Interface (CLI). For details, see the RealView Debugger v1.7 Command Line Reference Guide.
2.3 Configuring the ETM

Use the Configure ETM dialog to configure the ETM behavior. The options you configure in this dialog 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 2-21).

To access the Configure ETM dialog, 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.

Note

The Configure ETM dialog appears only when you are using an ETM-enabled processor variant.

The appearance of the Configure ETM dialog varies depending on the ETM architecture you are using. For architectures other than ETM v3, the Configure ETM dialog appears as shown in Figure 2-2 on page 2-12. The ETM architecture and protocol versions are displayed at the top of the Configure ETM dialog.
For ETM v3, the Configure ETM appears as shown in Figure 2-3.

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

Figure 2-3 The Configure ETM dialog for ETM v3
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 are dependent on 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 options depending on whether they are available on your target. In cases where a value, such as timestamping, is fixed due to a target-specific restriction, the value is displayed in the Configure ETM dialog, but the option 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.

The options in the Configure ETM dialog are:

- Trace data width
- Trace port mode on page 2-14
- Trace buffer packing on page 2-15
- FIFO overflow protection on page 2-16
- Trace coproc register transfer on page 2-17
- Enable Timestamping on page 2-18
- Cycle accurate tracing on page 2-18
- Suppress data on FIFO full on page 2-19
- Memory map decode on page 2-19
- ETM Pairing on page 2-19.

--- Note ---

Some of these options might be grayed out, depending on your system design.

2.3.1 Trace data width

The radio button selected, the default, shows the currently selected data port width. You can use this option to change the number of trace port pins used to broadcast trace information. 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.
2.3.2 Trace port mode

This control specifies the way the trace port is operated.

- For architectures other than ETM v3, 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 ETM v3, 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. Available options are:
  - **Dynamic** (for use with on-chip trace buffers)
  - **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 always enabled for ETMv3.
You can also use this section of the dialog to enable or disable half-rate clocking or suppress output from the trace port:

**Half-rate clocking enabled**

Select this option 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 option is not available when you are using ETM hardware implementing ETM v1.0. Hardware implementing ETM v1.1 might support the option, 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. Refer to the documentation that accompanies your hardware to see if it is available.

**Disable traceport**

Select this option 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.

### 2.3.3 Trace buffer packing

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

Options are:

**Automatic**

This mode allows the TPA to select the best packing mode for the current portwidth. You should 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 further reduces the resolution of timestamps.

Note
This option is only available for ETM targets that are connected using MultiTrace or RealView Trace. For MultiTrace and RealView Trace, Double Packing doubles the trace depth and Quad Packing quadruples it. For details of the trace depths supported by your TPA, refer to 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. Refer to the documentation for your TPA for further details.

2.3.4 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 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.

--- **Note** ---
This option is only available if you are using ETM v3, and is grayed out for all other ETM architectures.

--- **2.3.5 Trace coproc register transfer** ---
This area of the dialog 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. You can specify whether or not to trace data using the **Data tracing mode** option in the Analysis window **Edit** menu. (See **Data Tracing Mode** on page 2-54 for more information.)

--- **Note** ---
This option is only available if you are using ETM v3, and is grayed out for all other ETM architectures.
### 2.3.6 Enable Timestamping

Selecting this option enables the timestamp recording logic in the trace capture hardware. Timestamps are displayed in the **Time/unit** and **+Time** columns of the Analysis window (see *Column types* on page 2-60). You can change the format in which timestamps are displayed using the **Scale Time Units...** option in the **View** menu (see *Scale Time Units...* on page 2-57). 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.

If you want to view profiling information, you should enable either timestamping or cycle accurate tracing, or both. Better results in profiling are generally obtained with timestamping enabled. See *Viewing profiling information* on page 2-67 for information on the profile view.

Transmitting the timestamps uses noticeable 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. You should therefore only enable this feature when you have to.

——— **Note** ————

This feature is not supported by all types of trace capture hardware. Refer to the documentation that accompanies your hardware for more information.

### 2.3.7 Cycle accurate tracing

This option determines whether the ETM operates in cycle-accurate mode. When this option 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. The **Elem** column of the Analysis window 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 should enable either timestamping or cycle accurate tracing, or both. Better results in profiling are generally obtained with timestamping enabled.

When this option 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.

2.3.8 Suppress data on FIFO full

Selecting this option 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 option, the decompressor suppresses the data that the ETM might have traced incorrectly. However, some correctly traced data might also be suppressed.

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

2.3.9 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, refer to your system documentation.

2.3.10 ETM Pairing

This option should only be used when you are connecting to a dualcore platform and using the tracepoint multiplexer. It enables you to pair ETMs if you are connected to ETM hardware that contains multiple processors. Without ETM pairing in a multiprocessor system, the Trace Port Multiplexer (TPM) might return data to the incorrect ETM in some circumstances. When you specify the correct ETM pairing, RealView Debugger handles the switching of trace data streams automatically.

To configure ETM Pairing:

1. Click on the drop-down list and select the ETM that you want to pair with the current ETM, or No Pairing if you do not want to use ETM pairing.

2. If the current ETM is the master (the first ETM in the chain), check the Master ETM check box.
Note

This option is only available if you are connected to ETM hardware that contains multiple processors, and is grayed out otherwise.
2.4 Configuring Trace Capture

This section describes how to set simple and complex tracepoints from within the Code window, and how to start tracing after you have configured the conditions for trace capture. It contains the following sections:

- About setting tracepoints
- Setting simple tracepoints on page 2-22
- Setting complex tracepoints on page 2-29
- Editing tracepoints on page 2-40
- Starting trace capture on page 2-42.

2.4.1 About setting tracepoints

RealView Debugger enables you to set a variety of tracepoints to control the amount and content of program information that is traced. A tracepoint can be set on any of the following:

- a line or range of source code
- a line or range of assembly code
- a memory address or range of memory addresses.

After you have configured the details for trace capture and executed your program, the captured information is returned to the Analysis window. From there, you can perform further filtering by narrowing the results of the capture (see Filtering captured information on page 2-78).

To set simple or complex tracepoints, you must:

1. Ensure that you are connected to your target properly, and that your image is loaded into RealView Debugger.

2. Determine the area of interest within your program for which you want to collect trace information. Depending on the type of information you want to trace, you must give focus to your program within one of the following RealView Debugger Code window views:

   File Editor pane Src tab
   For basing your trace capture criteria on program instructions in source view.

   File Editor pane Dsm tab
   For basing your trace capture criteria on program instructions in disassembly view.

   Memory pane
   For basing your trace capture criteria on memory address ranges.
3. 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:

**Simple tracepoints**
For setting trigger points, trace start and end points, or trace ranges for memory and data accesses. See *Setting simple tracepoints* for details.

**Complex tracepoints**
For setting AND or OR conditions, counter conditions, and complex comparisons. These conditions can involve any supportable combination of trigger points and ranges. See *Setting complex tracepoints* on page 2-29 for details.

4. Select **View** → **Pane Views** → **Break/Tracepoints Pane** in the RealView Debugger Code window to display the Break/Tracepoints pane. This pane enables you to view, edit and track tracepoints. See *Editing tracepoints* on page 2-40 for more information on this pane.

You can now execute your program to begin trace capture with the details you have set, as described in *Starting trace capture* on page 2-42. See the section on *Controlling Execution* in the RealView Debugger v1.7 User Guide for details.

---
**Note**
The availability of tracepoint options is dependent 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.

### 2.4.2 Setting simple tracepoints

There are two options for setting simple tracepoints:

- If you want to specify a range within which to capture trace information, you can use the **Set Trace Range** option (see *Setting a simple tracepoint using Set Trace Range*).

- You can also set triggers, and trace start and end points. To do this, use the **Set/Toggle Trace Point** option (see *Setting a simple tracepoint using Set/Toggle Trace Point* on page 2-23).

**Setting a simple tracepoint using Set Trace Range**

To set a simple tracepoint using Set Trace Range:

1. In the File Editor pane, highlight a range of rows within your program, and right-click in the gray bar to the left of the source. A context menu is displayed.
2. Select **Set Trace Range** from the context menu.

RealView Debugger ensures that trace information is captured for only the range of program instructions you have selected. Any branches to memory and data values that are outside the range are not captured. These branches are represented as **Trace Pause** status lines in the Analysis window (see *Status lines* on page 2-64). To ensure that trace capture includes any data and memory accesses that might be branched to, you must use the **Trace Start Point** and **Trace End Point** options that are described in the following section.

--- **Note** ---

This option is not available with non-ETM targets.

---

**Setting a simple tracepoint using Set/Toggle Trace Point**

The Set/Toggle Trace Point List Selection dialog enables you to set trace ranges, trace start and end points, and triggers. These can be used individually or in conjunction with one another, to ensure capture of trace information for the precise area of interest.

**Trigger**

Set a trigger to indicate the area of interest in your program. The actual information to be traced depends on whether you have specified to collect trace before, after, or about the trigger (see the **Collect trace...** options in *Configuring trace options* on page 2-48).

**Trace range**

Set a trace range to indicate an area for which you want to capture 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* on page 2-64). You can set the following types of range:

- instructions only
- instructions and data.

You can also set an excluded trace range, to indicate an area for which trace information is not captured. You can set the following types of excluded range:

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

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

If you want to generate profile information, you should set trace start and end points (at the boundaries of the functions you want to profile) instead of setting ranges. In particular, setting an excluded trace range can lead to incorrect or misleading profiling data. See **Viewing profiling information** on page 2-67 for information on Profile view.

Trace start and end point

Set a trace start and end point to indicate an area for which you want to capture trace information. Trace information is returned from the specified start point until the specified end point, including any areas that are branched to. If you do not set a trace end point, trace information is returned from the start point onward.

Trace start and end points are not supported natively by ETM v1.1 or below. For these ETMs, 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 ETM v1.1 or below, and set three trace start points, the first option above is assumed. That is, you can set up to four start points, but are limited to two stop points.

When you set a tracepoint, such as a trigger, a corresponding **Clear** option becomes available in the List Selection dialog. 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

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

To set a simple tracepoint using the Set/Toggle Trace Point List Selection dialog:

1. In the RealView Debugger Code window, place the cursor on a row of interest within your program, or highlight a range of rows, and right-click on the gray bar to the left of the source. A context menu is displayed.
2. Select Set/Toggle Trace Point... from the context menu. This displays a List Selection dialog, as shown in Figure 2-4. From this dialog, you can select from a variety of options for setting a trigger, tracepoints or trace ranges.

![List Selection dialog for setting simple tracepoints](image)

**Note**

The options that appear in this dialog are dependent on the available resources and on the hardware 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.

3. Select the required option from the list. Options are:

**Set Trigger**

Sets an explicit trigger point on the selected address in the File Editor pane or memory pane. A trigger point enables you to capture trace before, after, or about the trigger point. See Trigger Mode on page 2-53 for information on selecting the trigger mode. When you select Set Trigger, an arrow is placed in the left margin, next to the line of code you have selected.

**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.

**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, and the amount of trace information returned is dependent on the buffer size that is currently set (see Configuring trace options on page 2-48).
If a trace range is set in conjunction with trace start and end points, then trace is only captured within that range. If the trace start point is before the start of the range, no trace information is captured until the start of the range. Similarly, if the trace end point is after the end of the range, no trace information is captured after the end of the range. In this case, trace start and end points specify that trace information is captured in the range specified only after it has passed a trace start point, and trace information is not captured after a trace end point has been passed.

This option is not available for connections that use the Simulator Broker.

**Trace Start Point (Instruction Only)**

Sets a trace start point on the selected address in the File Editor pane or memory pane. Tracing begins at the start point, and instructions only are traced. When you select **Trace Start Point (Instruction Only)**, an arrow  is placed in the left margin, next to the line of code you have selected.

--- **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, and the amount of trace information returned is dependent on the buffer size that is currently set (see *Configuring trace options* on page 2-48).

---

This option is only available for connections that use the Simulator Broker.

**Trace Start Point (Instruction and Data)**

Sets a trace start point on the selected address in the File Editor pane or memory pane. Tracing begins at the start point, and both instructions and data are traced. When you select **Trace Start Point (Instruction and Data)**, an arrow  is placed in the left margin, next to the line of code you have selected.

--- **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, and the amount of trace information returned is dependent on the buffer size that is currently set (see *Configuring trace options* on page 2-48).

---

This option is only available for connections that use the Simulator Broker.
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.

--- Note ---

Setting a trace end point but no trace start point results in different behavior depending on your system:

- If you are using an ETM-based system, no trace information is returned, and a warning is displayed in the **Cmd** tab of the **Output pane**.
- If you are using a simulator, tracing begins immediately and trace information is returned up until the trace end point is traced.

---

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

This option is not available for connections that use the Simulator Broker.

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, in this case **End of Trace Range (Instruction and Data)**, becomes available in the List Selection dialog. 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.

This option is not available for connections that use the Simulator Broker.
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 program except those within the range you specify.

Note

If the excluded range you specify contains a branch to another area of your program, 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, in this case End of Excluded Trace Range (Instruction and Data), becomes available in the List Selection dialog. 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.

This option is not available for connections that use the Simulator Broker.

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 program, 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, in this case End of Excluded Trace Range (Data Only), becomes available in the List Selection dialog. 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 \( \uparrow \) is placed in the left margin next to the line of code you have selected. This option is not available for connections that use the Simulator Broker.

### 2.4.3 Setting complex tracepoints

You can set complex tracepoints if you are using an ETM-based target. Select **Debug** → **Tracepoints** from the Code window main menu to display the **Tracepoints** menu, shown in Figure 2-5.

![Figure 2-5 Tracepoints menu](image)

From the **Tracepoints** menu, you can choose from the following options:

- Select **Address/Data** to set a complex tracepoint using the Set/Address/Data Break/Tracepoint dialog (see *Setting tracepoints with the Set Address/Data Break/Tracepoint dialog* on page 2-30)
- Use one of the four complex tracepoint dialogs to set commonly-used complex tracepoints (see *Using the complex tracepoint dialogs* on page 2-34).
Setting tracepoints with the Set Address/Data Break/Tracepoint dialog

Select Address/Data... from the Tracepoints menu to display the Set Address/Data Break/Tracepoint dialog, shown in Figure 2-6.

See the Breakpoints chapter in the RealView Debugger v1.7 User Guide for information on using this dialog. When trace support is enabled, the Set Address/Data Break/Tracepoint dialog contains the following additional tracepoint types:

**Trace 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).

**Trace Instr Fetch**

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

**Trace DataValue Read**

The address comparison is made against the source address of a data transfer cycle. The address compared is the address of the data value read, and not the address of the load instruction.
Trace DataValue Write

The address specified corresponds to the destination address of a data write instruction. The data write address is not the same as the address of the store instruction.

Trace DataValue Access

The address specified corresponds to access in either direction, read or write.

The HW Support area of the dialog is populated if you select a suitable tracepoint type and your current target supports the chosen type. Available types of hardware support are:

**HWPassCount**

The number of times a point can be passed over before the specified condition is enabled. Double-click on this item to display a Prompt dialog, shown in Figure 2-7.

![Figure 2-7 Setting the HWPass Count](image)

Enter the required pass count. If you do not set this item, it defaults to Off, and the condition is executed each time the point is hit.

**Match=Size of Data Access: (Any)**

Matches on the value of the data access size. The size of the data transfer is compared against the address you specify. Double-click on this item to display a List Selection dialog, shown in Figure 2-8 on page 2-32.
Figure 2-8  Setting the Data access size

Select the required value:

- Select Any if you do not want to match on size of data access. This is the default.
- Select Halfword if you want 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.
- Select Word if you want 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.

This option is only applicable if you have selected Trace DataValue Read, Trace DataValue Write, or Trace DataValue Access in the Break/Tracepoint Type section of the dialog.

Match=CheckCondition 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).

Double-click on this item to display a List Selection dialog, shown in Figure 2-9.

Figure 2-9 Setting the Condition Code
Select the required value:

- Select **Ignore** if you do not want to match on execution status. This is the default.
- Select **Pass** to match only instructions that executed.
- Select **Fail** to match only instructions that did not execute.

This option is only applicable if you have selected **Trace Instr Exec** in the Break/Tracepoint Type section of the dialog.

--- **Note** ---

If you are using ETM hardware implementing ETM version 1.0 or 1.1, this option is not applicable.

---

**OnBrk=Tracepoint Type**

Sets the type of tracepoint.

Double-click on this item to display a List Selection dialog, shown in Figure 2-10.

**Figure 2-10 Setting the Tracepoint Type**

Select the required tracepoint type:

- Select **Trigger** if you want to set an explicit trigger point on the selected address. This is the default.
- Select **Start Tracing** if you want to start tracing at the selected address.
- Select **Stop Tracing** if you want to stop tracing at the selected address.
- Select **Trace Instr** if you want to trace instructions only.
- Select **Trace Instr and Data** if you want to trace both instructions and data.
These options work in the same way as those in the List Selection dialog for setting a simple Tracepoint using Set/Toggle Tracepoint. See Setting a simple tracepoint using Set/Toggle Trace Point on page 2-23 for detailed information on these options.

**Chain=’then-prev’**

Indicates that the selected tracepoint is chained to another tracepoint to form a complex tracepoint. You can edit the tracepoint itself, but you cannot edit the chaining or the type of tracepoint.

**Using the complex tracepoint dialogs**

The complex tracepoint dialogs enable you to set specific types of complex tracepoint. In each dialog, you can select tracepoint types and conditions from the drop-down lists, and enter the required addresses or address ranges into the textboxes. An example of a complex tracepoint dialog is shown in Figure 2-11.

![Example of a complex tracepoint dialog](image)

**Figure 2-11 Example of a complex tracepoint dialog**

The following tracepoint types are available:

- **Trigger**
  - Sets an explicit trigger point on the selected address.

- **Trace Instr**
  - Traces instructions only.

- **Trace Instr and Data**
  - Traces both instructions and data.

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).

- **Instr Fetch**
  - The address of each instruction fetched is compared against the address you specify.
Data Access  The address of the data accessed, in either a read or write direction, is compared against the address you specify.

Data Read  The address of the data read from is compared against the address you specify.

Data Write  The address of the data written to is compared against the address you specify.

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

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

- Select the required item from a menu of items saved in your personal history file. To display this menu, click on the drop-down arrow. For example, choose from a browser, or select from your personal favorites list, or select from a list of previously-used addresses.

- Type the required address or address range 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 textbox is empty, RealView Debugger inserts start .. end into the textbox. Replace start with the start address and end with the end address.

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

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

  NOT Address Compare  When you select this option, RealView Debugger inserts $NOT$ into the textbox. Enter the required value.
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.

Value Mask

The value mask allows 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 textbox. Enter the value you want to compare against, and edit the mask to the required value.

NOT Value Compare

When you select this option, RealView Debugger inserts `$NOT$` into the textbox. Enter the required value.

--- Note

Not all options are available in all menus.
Available complex tracepoints are:
- **Trace on X after Y [and/or Z]...**
- **Trace on X after Y executed N times...** on page 2-38
- **Trace on X after A==B...** on page 2-38
- **Trace if A==B in X...** on page 2-39.

**Trace on X after Y [and/or Z]...**

Select **Trace on X after Y [and/or Z]...** from the Tracepoints menu to display the Trace on X after Y [and/or Z] dialog, shown in Figure 2-12.

![Figure 2-12 Trace on X after Y [and/or Z] dialog](image)

This dialog enables you to define a tracepoint that is only active when one or more specified conditions have been met. For example, you can set a tracepoint that is only active when the first specified function has been executed but the second has not.

To set the complex tracepoint:

1. From the first two drop-down lists, select the type of tracepoint that you want to set, for example **Trigger on Instr Exec**.

2. For the first tracepoint unit, specify the address or address range to test. The tracepoint unit triggers if the PC falls within the specified range.

3. For the second tracepoint unit, 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.

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

5. Click OK to set the tracepoint as specified.

**Trace on X after Y executed N times...**

Select Trace on X after Y executed N times... from the Tracepoints menu to display the Trace on X after Y executed N times... dialog, shown in Figure 2-13.

![Trace on X after Y executed N times dialog](image)

Figure 2-13 Trace on X after Y executed N times dialog

This dialog enables you to set a tracepoint that becomes active when the secondary condition has been met a specified number of times. See Setting up a complex tracepoint on page 2-105 for a detailed example.

To set the complex tracepoint:

1. From the first two drop-down lists, select the type of tracepoint that you want to set, for example Trigger on Instr Exec.

2. For the first tracepoint unit, specify the address or address range to test. The tracepoint unit triggers if the PC falls within the specified range.

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

4. For the second tracepoint unit, choose the tracepoint type and address range in the same way as for the first tracepoint unit.

5. Click OK to set the tracepoint as specified.

**Trace on X after A==B...**

Select Trace on X after A==B... from the Tracepoints menu to display the Trace on X after A==B dialog, shown in Figure 2-14 on page 2-39.
This dialog enables you to specify a tracepoint 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.

To set the complex tracepoint:

1. From the first two drop-down lists, select the type of tracepoint that you want to set, for example Trigger on Instr Exec.

2. For the first tracepoint unit, specify the address or address range to test. The tracepoint unit triggers if the PC falls within the specified range.

3. Set the second tracepoint unit:
   - Select the tracepoint type, for example Data Access.
   - Specify the address, address range, or variable to test.
   - Specify the value you want to compare the address, address range, or variable against.

4. Click OK to set the tracepoint as specified.

**Trace if A==B in X...**

Select Trace if A==B in X... from the Tracepoints menu to display the Trace if A==B in X dialog, shown in Figure 2-15.
This dialog enables you to set a tracepoint 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.

To set the complex tracepoint:

1. From the first two drop-down lists, select the type of tracepoint that you want to set, for example **Trigger** on **Instr Exec**.

2. Set the first tracepoint unit:
   - Select the tracepoint type, for example **Data Access**.
   - Specify the address, address range, or variable to test.
   - Specify the value you want to compare the address, address range, or variable against.

3. Set the second tracepoint unit:
   - Select the tracepoint type, for example **Data Access**.
   - Specify the address or address range to test.

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

### 2.4.4 Editing tracepoints

Whenever you set a tracepoint of any type, it is displayed in the Break/Tracepoints pane, shown in Figure 2-16.

![Figure 2-16 The Break/Tracepoints pane](image)

---

**Note**

A complex tracepoint might be displayed as two or more tracepoints in the Break/Tracepoints pane.
To see the address and command details for a specific tracepoint, expand the display by clicking ▷.

From the Break/Tracepoints pane, you can:
- Modify tracepoints
- Disable tracepoints
- Clear tracepoints on page 2-42
- Find corresponding source code on page 2-42.

For general details on using this pane, including how to hide or display it, see the Breakpoints chapter in the RealView Debugger v1.7 User Guide.

Modify tracepoints

To modify a tracepoint, right-click on the required tracepoint in the Break/Tracepoints pane and select Edit Break/Tracepoint... from the context menu. This displays the Set Address/Data Break/Tracepoint dialog, shown in Figure 2-6 on page 2-30. The Class field shows Tracepoint to show that you are editing a tracepoint and not, for example, a breakpoint. Edit the tracepoint as required, then click OK to confirm any changes and close the Set Address/Data Break/Tracepoint dialog.

Note

Only tracepoints set using an ETM-based target can be modified. If you attempt to edit tracepoints set using a simulator, a warning is displayed, telling you to delete the existing tracepoint and set a new one.

See the Breakpoints chapter in the RealView Debugger v1.7 User Guide for more information on using the Set Address/Data Break/Tracepoint dialog.

Note

Tracepoints that succeed other tracepoints in a chain display the value Chain="then-prev" in the HW Support area of the Set Address/Data Break/Tracepoint dialog. These tracepoints cannot be edited.

Disable 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 complex 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 complex 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.

Clear tracepoints

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

If you want to clear a complex tracepoint, you must clear its components in the reverse order of that in which they were set, that is, from the bottom up.

If you clear a tracepoint that is chained with others to form a complex 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.

Find corresponding source code

To locate the line of source code corresponding to a tracepoint, right-click on the entry and select Show Code from the context menu. In this case, an arrow ➔ is placed next to the line of source code in the File Editor pane to which the tracepoint corresponds.

2.4.5 Starting trace capture

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

1. Display the Analysis Window by selecting View → Analysis Window from within a Code window.

2. Ensure that Tracing Enabled is selected in the Edit menu of the Analysis window (see Configuring trace options on page 2-48).

    ———— 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. Execute your program by selecting **Debug → Execution Control → Go (Start Execution)**.
2.5 Using the Analysis window

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

- Overview of the Analysis window
- Configuring view options on page 2-56
- Configuring trace options on page 2-48
- Interpreting the data on page 2-58
- Viewing profiling information on page 2-67
- Finding information on page 2-72
- Filtering captured information on page 2-78
- Changing the display on page 2-86
- Saving and loading trace information on page 2-90
- Other window elements on page 2-92.

2.5.1 Overview of the Analysis window

The Analysis window enables you to:

- view the current state of the trace
- configure global tracing options
- analyze the results of a trace capture to give profiling information
- view tracing and profiling information, using one of seven tabs (see Tabbed view types on page 2-59)
- filter the results of a trace capture
- search for a specific item of trace information
- manipulate the display of trace information.

You can display the Analysis window from the Code window in the following ways:

- select Analysis Window from the View menu (the Raw tab view is displayed by default)
- select Tools → Analyzer/Trace Control → Display Profile View... (the Profile tab is displayed).

Figure 2-17 on page 2-45 shows an example of the Analysis window.
Analysis window toolbar

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

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.

Save Trace Buffer to File...

This option enables you to save the current trace information to a file. See Save Trace Buffer to File... on page 2-90 for information on using this dialog.

Copy

Copies the selected text to the clipboard.

Enable Trace

This option globally enables tracing, and is selected by default. See Tracing Enabled on page 2-49 for more information.
Track in Code Window

When this option is selected, 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.

Change to Next Active Connection

This button, and its associated menu, enables you to change the current connection and to manage windows attachment during your debugging sessions. Click this button to switch to the next available active connection. Click on the drop-down arrow to display the list of active connections. The current connection is marked with an asterisk, as shown in Figure 2-18.

Figure 2-18 Connection menu

If the Code window is attached, a check mark is displayed next to the connection name.

For detailed information on using this feature, see Using the Connection menu on page 5-20 in Chapter 5 Working with Multiple Target Connections.

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.
2.5.2 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 2-49
- Tracing Enabled on page 2-49
- Configure Analyzer Properties... on page 2-49
- Select Analysis Configuration... on page 2-50
- Set Trace Buffer Size... on page 2-50
- Set Amount of Trace Buffer to Read... on page 2-51
- Store Control-flow Changes Only on page 2-52
- Buffer Full Mode on page 2-52
- Trigger Mode on page 2-53
- Data Tracing Mode on page 2-54
- Automatic Tracing Mode on page 2-54
- Set/Edit Event Triggers on page 2-55
- Clear All Event Triggers on page 2-55
- Physical to Logical Address Mapping... on page 2-55.

Copy

This option copies the selected text to the clipboard.

Connect Analyzer

Connects to an analyzer.
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. If you are using Multi-Trace, this flag is set automatically. If you are using RealView Trace, you must configure it manually. See the chapter on Using RealView Trace in the RealView ICE User Guide for information on how to do this.

--- Note ---

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 program stops executing.

If you stop tracing during program 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 program execution.

--- Note ---

This option is enabled only for ETM-based targets.

Configure Analyzer Properties...

This option enables you to configure the analyzer settings. If your target is ETM-enabled, this option displays the Configure ETM dialog. If you are using a DSP Group processor, this option opens a List Selection Dialog, shown in Figure 2-19 on page 2-50.
Select the required option from the list. Options are:

**Stop-Collect-Run profiling (intrusive)**

This option instructs RealView Debugger to store all the trace data for a session. When the on-chip trace buffer is full, the target processor is stopped and the trace data is collected. The on-chip trace buffer is cleared, and tracing continues until it is full again. The new trace data is appended to the RealView Debugger trace buffer. This option enables you to collect more trace data, but is intrusive, because it involves stopping and starting the target processor.

**Default Ring-buffered last N traces**

This option instructs RealView Debugger to store only the last N traces in a session. When the trace buffer becomes full, older information is cleared from the buffer as new information enters it. This option is non-intrusive, but only a single buffer is collected.

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

**Select Analysis Configuration...**

This option displays a dialog enabling you to select the analyzer configuration.

**Set Trace Buffer Size...**

This option enables you to set the size of the trace buffer. When you select this option, a Prompt dialog is displayed, as shown in Figure 2-20 on page 2-51.
Figure 2-20 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. Refer to the documentation that accompanies your analyzer for details.

---

Set Amount of Trace Buffer to Read...

This option enables you to limit the range of the trace buffer to be displayed in the Analysis window to a specified size or range. This is useful if you have captured a large trace buffer, and you want to view only a small section at a time.

--- Note ---

This option does not limit the amount of data captured. It limits only the amount of trace data read back from the trace buffer.

---

When you select this option, a Prompt dialog is displayed, as shown in Figure 2-21.

---

Figure 2-21 Setting the amount of trace buffer to read
Enter any of the following, then click **Amount**:

**Number**
Specify the maximum number of entries, from the start of the trace buffer, you want to be displayed, either in decimal or in hexadecimal, such as 1024 or 0x400.

**0**
Enter 0 if you want the default number of entries to be displayed. The default value varies depending on the number of entries your target can support. For ETM-based targets, the value of the entire buffer is used.

**Number .. Number**
Enter a range to indicate the number of entries before and after the trigger that you want to be displayed, such as -50 .. 50 to display 50 entries before, and 50 entries after, the trigger point.

**Store Control-flow Changes Only**
This option enables you to capture and return control-flow information (branches). It ensures that only control-flow branches are stored in the trace buffer. This reduces the amount of information stored in the buffer during a trace capture session.

This option is especially useful when you are interested in using the **Profile** tab to analyze branching information only.

This option is not available for ETM targets, because for these targets it is set by default and cannot be changed.

**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.

If you are using an ETM target, you cannot deselect this option.

Note

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

Note

This option is not available for ETM targets, as Continue Collecting on Buffer Full is set by default for these targets and cannot be changed.

Trigger Mode

Enables you to configure whether trace data is collected before, around or after the trigger. 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 data prior to the trigger position. For example, if your buffer size is set to 50, the 50 rows of trace information directly preceding the trigger point are returned.

Collect Trace Around Trigger

This option instructs RealView Debugger to capture all data around the trigger position. In this case, half of the data prior to reaching the trigger, and half of the data after reaching the trigger, is captured.

Collect Trace After Trigger

This option instructs RealView Debugger to capture all data after the trigger position. For example, if your buffer size is set to 50, the 50 rows of trace information directly following the trigger point are returned.

Stop Processor on Trigger

This option instructs RealView Debugger to stop the target processor when the trigger is hit.
—— Note ————
This option is enabled only for ETM-based targets.

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 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 2-21 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 program to generate trace information. When you stop the processor, the trace information currently in the buffer is returned to the Analysis window.

—— Note ————
When a tracepoint is set, Automatic Trace 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, RealView Debugger performs tracing using any tracepoints that you have set.

**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.

--- **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.
2.5.3 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 Interleaved Source**
- **Show Details View** on page 2-57
- **Show Position Relative to Trigger** on page 2-57
- **Scale Time Units...** on page 2-57
- **Define Processor Speed for Scaling...** on page 2-58
- **Automatic Update on New Buffer** on page 2-58
- **Automatic Update on Append to Buffer** on page 2-58
- **Append New Buffer to Existing** on page 2-58.

**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 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.
Show Details View
Displays status-bar information in the details view at the bottom of the window (see Other window elements on page 2-92). 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 is displayed. Select one of the following time formats, then click OK:

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

Note
The default format is dependent 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... on page 2-58 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, in order to set the scaling between cycles and timestamp values in the current buffer, as shown in the \textit{Time/cycl} column. When you select this option, a Prompt dialog is displayed, as shown in Figure 2-22.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure22.png}
\caption{Defining processor speed}
\end{figure}

Enter a clock speed in MHz, then click Speed.

\section*{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.

\section*{Automatic Update on Append to Buffer}

This option instructs the Analysis window to keep its existing contents, and new information to be appended when a new buffer load of trace data is returned.

\section*{Append New Buffer to Existing}

This option enables new trace buffers to be appended to the existing trace buffer. The indexes of the trace information are prefixed with a number corresponding to the buffer in which they were captured. For example, an index value of 2.555 indicates record 555 in the second stored buffer load.

This option is grayed out if a trace buffer does not already exist.

\subsection*{2.5.4 Interpreting the data}

This section describes the ways in which you can use the Analysis window to interpret data that has been returned as the result of a trace capture you have performed. It contains the following sections:

\begin{itemize}
\item \textit{Tabbed view types} on page 2-59
\item \textit{Column types} on page 2-60
\end{itemize}
Tabbed view types

Figure 2-23 shows the tabs available at the bottom of the Analysis window.

The tabs are:

- **Raw** Displays the symbolic representation of traced instructions and data, interleaved.
- **Code** Displays the symbolic representation of instructions that have been traced.
- **Data** Displays the symbolic representation of data that has been traced, if you have specified data capture in the trace configuration. See Configuring trace capture on page 2-21.
- **Dsm** Displays the disassembly of traced instructions and data.
- **Source** Displays the source code preceded by a single line giving the file name, line number(s) and function that each block relates to.
- **Func** Displays the branches to functions.
- **Profile** Displays a statistical analysis of your trace information. You can access this view by clicking the Profile tab, or by selecting Tools → Analyzer/Trace Control → Display Profile View... from the Code window.

The Profile tab enables you to perform various profiling analyses of the trace data. 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. For detailed information on viewing profiling information, see Viewing profiling information on page 2-67.

Status lines on page 2-64.
Column types

The Analysis window displays tracing information as a table of values, where the significance of the contents depends on the currently displayed tab. When the Profile tab is selected, profile view columns are displayed. When any other tab is selected, trace view columns are displayed. You can set which columns are displayed for each view using the options in the Profiling data and Trace columns menus. See Changing the display on page 2-86 for information on specifying which columns are displayed.

Available trace view columns are:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Elem</strong></td>
<td>Displays the position of each element within the trace buffer, where the value can represent either:</td>
</tr>
<tr>
<td></td>
<td>• An index within the trace buffer.</td>
</tr>
<tr>
<td></td>
<td>• A cycle number, if your trace capture device supports cycle-accurate tracing. If you select this option, you must also ensure that the option <strong>Cycle accurate tracing</strong> is selected in the Configure ETM dialog (see Configuring the ETM on page 2-11).</td>
</tr>
<tr>
<td><strong>Time/cycl</strong></td>
<td>Displays the timestamp value. You can change the format of the values by selecting Scale Time Units from the View menu.</td>
</tr>
<tr>
<td><strong>+Time</strong></td>
<td>Displays a delta timestamp value, indicating the time taken between the previous instruction and the current instruction. You can change the format of the values by selecting Scale Time Units from the View menu.</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Displays the access type of the current element, which can be any of:</td>
</tr>
<tr>
<td></td>
<td>• Bus</td>
</tr>
<tr>
<td></td>
<td>• Code</td>
</tr>
<tr>
<td></td>
<td>• Data</td>
</tr>
<tr>
<td></td>
<td>• DMA</td>
</tr>
<tr>
<td></td>
<td>• Exec</td>
</tr>
</tbody>
</table>
### Int
- Interrupt vectoring

### No Exec
- Instruction was not executed

### Pin
- Pin state change

### Pref
- Prefetch (so not executed)

### Prob
- External probe state change.

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

This column is displayed only when **Access Type** is selected in the **Trace columns** menu.

### Address
- Displays the address of the instruction or data accessed.

This column is displayed only when **Address as Value** is selected in the **Trace columns** menu.

### Symbolic
- Displays the symbolic information for the current range, and takes the form `symbol`.

This column is displayed only when **Address as Symbol/Line** is selected in the **Trace columns** menu.

### Count
- For the **Func** tab, this shows the number of times that the function was entered and exited. For all other tabs, it shows the number of times a particular address was accessed.

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

This column is displayed only when **Count of Hits** is selected in the **Trace columns** menu.

**Note**
- Generating the Count column can be slow, and consumes a large amount of memory.

### Other
- Shows the address, data value, and disassembly information for each particular instruction.

This column is displayed only when **Interpretation of Data** is selected in the **Trace columns** menu.
Available profile view columns are:

**1st**
Displays the element number representing the first access to each function.

This column is displayed only when **First Instance** is selected in the **Profiling data** menu.

**Exec%**
Displays the percentage of time taken to execute a particular function, where the entire trace buffer represents 100%. This represents the PC in the range of the function itself, that is, where it does not branch off to a subroutine call.

This column is displayed only when **Time% In Self** is selected in the **Profiling data** menu.

**B=>E%**
Displays the percentage of time spent from entry to exit of a particular function, including all children.

This column is displayed only when **Time% Including Children (B=>E)** is selected in the **Profiling data** menu.

**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 displayed only when **Range Types** is selected in the **Profiling data** menu.

**Address**
Displays the address of the instruction or data accessed.

This column is displayed only when **Address as Value** is selected in the **Profiling data** menu.

**Symbolic**
Displays the symbolic position information for the current element, and takes the form \( \text{symbol} + \text{offset} \). For example, \( \text{Arr}_2 \_\text{Glob} + 0x65 \) might be a data access to the variable address \( \text{Arr}_2 \_\text{Glob} \), with an offset of 0x65.

Alternatively, this can take the form \( \text{source_module_name}\#\text{line_number} \). For example, \( \text{MAIN}\#144 \) might be an access at line 144 that is within the file \text{main}(). The \( \text{source_module_name} \) can be any symbolic information, including a function, module, variable, or low-level symbol.

This column is displayed only when **Range Symbols** is selected in the **Profiling data** menu.

**Exec/B=>E/B=>E Avg**
Available only in the **Profile** tab. Displays the following information:
- Exec, the total time spent in execution(s) of this function
- B=>E, the total beginning to end time of all executions of this function
- B=>E Avg, the average of all the individual times spent on the execution of this function.

This column is displayed only when Exec/B=>E/B=>E Avg is selected in the Profiling data menu.

**Count**

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

--- Note ---

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

---

This column is displayed only when Count of Calls is selected in the Profiling data menu.

**Min/Max B =>E**

Displays the minimum time and 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 lowest and 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 2-57).

This column is displayed only when Min/Max Times is selected in the Profiling data menu.

**Histogram**

Displays the Histogram column. The histogram is displayed as a pink bar. You can view the histogram as a linear or a logarithmic function. For information on selecting the type of histogram, see Data view options on page 2-70.

This column is displayed only when Histogram View is selected in the Profiling data menu.
Status lines

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 can display any of the following messages:

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, 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: Trace branch address does not match instruction's branch address

RealView Debugger has branch addresses from both the trace and from the image loaded into the debugger, and these addresses do not match. This error is uncommon, and only occurs if you are using an XScale target.

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 will be 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 stop 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 should have been 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 program 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 program is still running, and the program attempts to execute code outside the loaded image.

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 program 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 bytecode state, trace data ignored
The ETM detected the processor entering bytecode state. The decompressor is unable to decompress bytecode execution, so all trace output is suppressed until the processor leaves bytecode 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: No data in trace buffer
The trace buffer is composed entirely of zero. This warning is very rare, and only occurs if you are using an XScale target.
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: Too many checkpoints in XScale trace buffer

Indicates that more than two checkpointed entries were found in the buffer. The decompressor has attempted to use the most recent entries. This message only occurs when you are using an XScale target.

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.

2.5.5 Viewing profiling information

The Profile tab in the Analysis window enables you 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, you should 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 program have been traced when interpreting profile information.

You can access this view by clicking the Profile tab, or by selecting Tools → Analyzer/Trace Control → Display Profile View... from the Code window. An example of Profile View is shown in Figure 2-24 on page 2-68.
Figure 2-24 Example of Profile View

You can control the information that is shown in the Profile tab using the Profiling data menu. This menu contains three sets of options:

- **Column options**
- **Data view options** on page 2-70
- **Data interpretation options** on page 2-71.

**Column options**

This section of the Profiling data menu enables you to select the columns that are displayed in the Profile tab. It contains the following options:

- **First Instance**
  - Displays the 1st column. This shows the position of the first instance of the function in the buffer. This option is disabled by default.

- **Time% In Self**
  - Displays the Exec% column. This shows, as a percentage of the whole, the time spent in the range of this function. This does not include time spent in the descendents of the function. This option is enabled by default.

- **Time% Including Children (B=>E)**
  - Displays the B=>E% column. This shows, 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 option is enabled by default.
Range Types
Displays the Type column. This shows the type of range, which is usually Func. Where there are no functions, for example if the code is in assembly language, the range type is Module. This option is disabled by default.

Address as Value
Displays the Address column. This shows the address of the instruction or data accessed. This option is disabled by default.

Range Symbol
Displays the Symbolic column. This shows the symbolic information for the range. This option is enabled by default.

Exec/B=>E/B=>E Average
Displays the Exec/B=>E/B=>E Avg column. This shows the absolute time in the range, the beginning to end of the range, and the average time from beginning to end. This option is disabled by default.

Count of Calls
Displays the Count column. This shows the number of calls to the range. This option is enabled by default.

Min/Max Times
Displays the Min/Max B=>E column. This shows the minimum time and 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 lowest and 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 2-57). This option is disabled by default.

Histogram View
Displays the Histogram column. The histogram is displayed as a pink bar. You can view the histogram as a linear or a logarithmic function. For information on selecting the type of histogram, see Data view options on page 2-70.
This option is enabled by default.

——— Note ————
If you are using a simulator, some columns might not be available. Columns that are not available are grayed out in the menu.
Data view options

This section of the Profiling data menu enables you to specify the call-graph information displayed. It contains the following options:

Parents of Function

Displays the parents of the function. A parent is a function that makes a call to the function being profiled. Parents are displayed before the function line, and the Symbolic information is indented. Histograms, if present, are a light gray color.

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. Children are displayed after the function line, and the Symbolic information is indented. Histograms, if present, are a dark gray color.

This option is enabled by default.

Figure 2-25 on page 2-71 shows an example of call-graph information displayed in the Profile tab.
Figure 2-25 Example of call-graph information

Parents and children are displayed for each function. Parents are displayed above the function line and indented by two spaces. Children are displayed below the function line and indented by two spaces. The groups of functions are separated by a ruler line. The ruler line is not present if neither parents nor children are displayed.

See Capturing profiling information on page 2-100 for a detailed example of how you can use RealView Debugger to capture profiling information.

Data interpretation options

This section of the Profiling data menu enables you to control the way that data is interpreted for display. It contains the following options:

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.
Relative %ages

This set of options enables you to specify the value that time percentages are calculated relative to. Select one of the following:

**Parent/Child %ages Relative to Whole Time**

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.

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

The values shown for Exec% and B=>E% for parents and children are shown relative to the absolute B=>E time of the function.

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

Displays Time% values for parents and children relative to the parent/child. For parents, the values for Exec% and B=>E% are shown relative to the total B=>E time for all instances of the parent function. For children, the values for Exec% and B=>E% are shown relative to the total B=>E time for all instances of the child function.

These options are grayed out if at least one of Parents of Function or Children of Function is not selected.

### 2.5.6 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 complete menu options are as follows:

- **Find Trigger**
- **Find Position Match...** on page 2-73
- **Find Time Match...** on page 2-73
- **Find Raw Address Match...** on page 2-74
- **Find Address Expression Match...** on page 2-75
- **Find Data Value Match...** on page 2-76
- **Find Symbol Name Match...** on page 2-77
- **Find Next Match** on page 2-78
- **Find Previous Match** on page 2-78.

**Find Trigger**

Locates the row of trace output representing the trigger point within your code. For details on setting triggers, see **Setting simple tracepoints** on page 2-22.
Find Position Match...

Enables you to locate a specific element number within the trace buffer (see the description of the **Elem column** in *Column types* on page 2-60). When you select this option, a Prompt dialog is displayed, as shown in Figure 2-26.

![Figure 2-26 Finding a position match](image)

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 is displayed, as shown in Figure 2-27 on page 2-74.
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 2-57 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 2-57 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 is displayed, as shown in Figure 2-28 on page 2-75.
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 is displayed, as shown in Figure 2-29 on page 2-76.
Figure 2-29 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 is displayed, as shown in Figure 2-30 on page 2-77.
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 is displayed, as shown in Figure 2-31.

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.

**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.

2.5.7 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. A total of 16 filters can be set.

The complete menu options are:

- Filter on Position Match...
- Filter on Time Match... on page 2-79
- Filter on Raw Address Match... on page 2-80
- Filter on Address Expression Match... on page 2-81
- Filter on Data Value Match... on page 2-82
- Filter on Symbol Name Match... on page 2-83
- Filter on Access Type Match... on page 2-84
- Filter on Percent Time Match... on page 2-85
- AND Filters (vs. OR) on page 2-85
- Clear Filtering on page 2-86.

**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 Column types on page 2-60). When you select this option, a Prompt dialog is displayed, as shown in Figure 2-32 on page 2-79.
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 **Timelimit** column. Additionally, this affects values in the **Exec%** column in the **Profile** tab. When you select this option, a Prompt dialog is displayed, as shown in Figure 2-33.
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 2-57 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 2-57 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 is displayed, as shown in Figure 2-34.

![Prompt dialog](image)

**Figure 2-34 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 is displayed, as shown in Figure 2-35. An address expression can be a function, structure, or array symbol.

![Figure 2-35 Filtering on an address expression match](image-url)
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 is displayed, as shown in Figure 2-36.

![Figure 2-36 Filtering on a data value match](image)

**Figure 2-36 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 is displayed, as shown in Figure 2-37.

![Figure 2-37 Filtering on a symbol name match](image)

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 is displayed, as shown in Figure 2-38.

![Figure 2-38 Filtering on an access type match](image)

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 are those supported by RealView Debugger, but not all of the options listed are supported for tracing by all targets. For example, the 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.
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 is displayed, as shown in Figure 2-39.

![Figure 2-39 Filtering on a percent time match](image)

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 halfway through execution.

**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.

### 2.5.8 Changing the display

This section describes the **Trace columns**, **Profiling data**, and **Sort** menu options you can use to customize the appearance of the Analysis window display.

**Trace columns and Profiling data menus**

The options in these menus enable you to indicate the columns you want to display or hide. To display a particular column, select it (✓) in the **Trace columns** or **Profiling data** menu. To hide the column, deselect the menu entry.

---

**Note**

The availability of columns is dependent on the tab that is currently displayed.

---

For a description of the type of information contained in each column of the Analysis window, see **Column types** on page 2-60.

The options in the **Trace columns** menu control what is displayed in all tabs except **Profile**. These options are:

- **Position**
  
  Displays the **Elem** column.

- **Absolute Time**
  
  Displays the **Time/unit** column.

- **Relative Time**
  
  Displays the **+Time** column.
Access Type  Displays the Type column.

Address as Value  
Displays the Address column.

Address as Symbol/Line  
Displays the Symbolic column.

Data Value in Hex  
Displays all values in the Time/unit and +Time columns in hexadecimal format.

Data Value in Decimal  
Displays all values in the Time/unit and +Time columns in decimal format.

Interpretation of Data  
Displays the Other column.

Count of Hits  
Displays the Count column.

The options in the Profiling data menu control what is displayed in the Profile tab. These options are:

First Instance  
Displays the 1st column.

Time% In Self  
Displays the Exec% column. This column is displayed only when Time% In Self is selected in the Profiling data menu.

Time% Including Children (B=>E)  
Displays the B=>E% column.

Range Types  
Displays the Type column.

Address as Value  
Displays the Address column.
Range Symbols
Displays the Symbolic column.

Exec/B=>E/B=>E Avg
Displays the Exec/B=>E/B=>E Avg column.

Count of Calls
Displays the Count column.

Min/Max Times
Displays the Min/Max B=>E column.

Histogram View
Displays the Histogram column.

Parents of Function
Displays the parents of the function

Children of Function
Displays the children of the function

Use Logarithmic Scale For Histogram
Displays the histogram bar using a logarithmic function of the Exec% or B=>E value instead of a linear function.

Parent/Child %ages Relative to Whole Time
Displays the values of Exec% and B=>E% for parents and children as a percentage of the whole time traced.

Parent/Child %ages Relative to Function B=>E
Displays the values of Exec% and B=>E% for parents and children relative to the absolute B=>E time of the function.

Parent/Child %ages Relative to Parent/Child B=>E
Displays Time% values for parents and children relative to the parent/child.

See Viewing profiling information on page 2-67 for detailed information on using Profile view.
Sort menu

The options in this menu enable you to sort the trace information 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 checked. If you click on the column header again to reverse the order, the item **Reverse Sort** is automatically checked.

**Note**

Sorting large traces can be slow, and consumes a large amount of memory.

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

**By Time/Entry**

Sorts the output by the **Exec%** column in the **Profile** tab, and by the **Time/unit** column in all other tabs. This is the default sort option.

**By Address**

Sorts the output by the **Address** column in all tabs.

**By Name**

Sorts the output by the **Symbolic** column in all tabs.

**By Data Value**

Sorts the output by either of the **Data** columns (decimal or hexadecimal). Not available in the **Profile** tab.

**By Access Type**

Sorts the output by the **Type** column in all tabs.

**By Relative Time**

Sorts the output by the **B=>E%** column in the **Profile** tab, and by the **+Time** column in all other tabs.

**By Count**

Sorts the output by the **Count** column in all tabs.
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.

2.5.9 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 2-91
- Close Loaded File on page 2-91
- Print Trace Lines... on page 2-92
- Connection on page 2-92
- Exit Window on page 2-92.

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:

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 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 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 *.

See Chapter 5 Working with Multiple Target Connections for detailed information on managing multiple connections.

Exit Window

This option closes the Analysis window.

2.5.10 Other window elements

This section describes the following additional elements of the Analysis window:

• Details view
• Context menu on page 2-93.

Details view

The details view is located directly above the tabs at the bottom of the Analysis window, as shown in Figure 2-40. To display status-bar information, select the option Show Details View from the View menu.

Figure 2-40 Details view
The details view provides different information depending on the tab currently displayed:

**Amount of data**

The amount of data in the trace buffer. This value is contained in square brackets.

**Address**

The target memory address of the currently selected line.

**Data**

The data value, if any, associated with the currently selected line.

**Disassembly**

A disassembly of instructions at the currently selected line.

**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:

**Source 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.

---

**Disassembly 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.

---
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 is displayed (Figure 2-41).

![Figure 2-41 Time measurement dialog](image)

**Time Measure from Trigger**

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

**Find Next**

Searches the trace output for the next instance of the search item you have specified. See *Finding information* on page 2-72 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 2-72 for details on performing searches in the Analysis window.

**Copy**

Copies the selected text to the clipboard.
2.6 **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 program. It contains the following sections:

- **Introduction to the examples**
- **Finding the cause of a data abort** on page 2-96
- **Capturing profiling information** on page 2-100
- **Setting up a complex tracepoint** on page 2-105.

2.6.1 **Introduction to the examples**

It is recommended that you perform these examples before using trace on your own programs, 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 located in the `install_directory\RVDS\Examples\...` directory:

- `primes.axf`
- `dhrystone.axf`.

The source file components are located a directory level above the images, that is, in the `...\primes` and `...\dhrystone` directories.

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 Multi-ICE or RealView ICE
- trace capture hardware, such as Multi-Trace or 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 2-96
- **Capturing profiling information** on page 2-100
- **Setting up a complex tracepoint** on page 2-105.
2.6.2 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 program.

The primes program used in this example is designed to calculate the \( n \)th prime number, where you are prompted to indicate \( n \) when running the program. However, execution of the program results in a data abort.

To perform this example and determine the cause of the data abort:

1. Load the example image primes.axf into the debugger. This file is located in the \Examples directory in your root installation. The tab for primes.cpp is displayed in the File Editor pane.

2. Enable data aborts in the vector catch. This procedure varies depending on the JTAG interface unit that you are using:
   - If you are using Multi-ICE:
     1. In the Register pane of the Code window, click the Debug tab. Debugger internal values are displayed.
     2. Double-click on the vector_catch value, and change the value to 0x13B.
   - If you are using RealView ICE:
     1. Select Simple Breakpoints → ProcessorEvents... from the Debug menu of the RealView Debugger Code window to open the Processor Events List Selection dialog.
     2. Ensure that data abort is checked in the list of processor events.
   - If you are using a non-ARM JTAG interface unit, refer to the documentation accompanying your hardware for information on how to enable data aborts.

This enables data abort exceptions to be caught by the debugger.

3. Configure tracing options as follows:
   a. Select Analysis Window from the View menu of the RealView Debugger Code window. The Analysis window is displayed.
   b. Select Trigger Mode → Collect Trace Before Trigger from the Edit menu of the Analysis window.
   c. Select Trigger Mode → Stop Processor On Trigger from the Edit menu of the Analysis window.
d. Select Configure Analyzer Properties... from the Edit menu of the Analysis window to display the Configure ETM dialog. In the Data tracing mode section of the dialog, ensure that Trace data and address is selected. In the Trace data width section of the dialog, ensure that 16 bit is selected. Click OK to close the Configure ETM dialog.

This ensures that a buffer-load of trace data is captured for the area of your program 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 program where the data abort occurs.

4. 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 program.
   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 is displayed.
   d. In the dialog, enter the value 0x10 and click Set. This displays the region of memory, the data abort vector in this case, for which a trigger must be set. An arrow is placed next to the address 0x10.

5. Set a trigger point as follows:
   a. In the Dsm tab, right-click in the left margin at the address 0x10. A context menu is displayed.
   b. Select Set/Toggle Trace Point... from the context menu. A List Selection dialog is displayed.
   c. In the List Selection dialog, 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 2-43 on page 2-99.
6. Set a trace range to capture instruction and data accesses:
   a. Right-click in the left margin at the address 0x0. A context menu is displayed.
   b. Select **Set/Toggle Trace Point...** from the context menu. A List Selection dialog is displayed.
   c. In the List Selection dialog, 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 2-99. The end of the trace range is automatically set to the end of memory space (0xFFFFFFFF).
7. Execute the image by selecting **Debug → Execution Control → Go (Start Execution)** from the Code window. 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 (ensuring that the **StdIO** tab is selected), enter 20 and press Enter. As a result of an error in the program, a data abort occurs. Because you have set a trigger point at the data abort address, the results of the trace capture are returned.

9. To view the results of the trace capture, select **Analysis Window** from the **View** menu of the RealView Debugger Code window. The Analysis window is opened, and the **Raw** tab is displayed.

10. In the Analysis window, click the **Dsm** tab at the bottom to see the disassembly of program instructions, as shown in Figure 2-44 on page 2-100.
11. Select **Find Trigger** from the **Find** menu of the Analysis window. RealView Debugger locates the row in the output representing the trigger point you have set. In the rows directly above the trigger point, you can see that a value of -1 (0xFFFFFFFF) is being used as a pointer. 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 program 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.

### 2.6.3 Capturing profiling information

This example demonstrates how you can use RealView Debugger to capture profiling information for your program.

The dhrystone program 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 program. In this example, assume that you want to analyze the execution times of all functions that are executed in the main for loop that is run repeatedly.
To perform this example and capture profiling information:

1. Load the example image dhrystone.axf into the debugger. This file is located in the `install_directory\RVDS\Examples\...` directory. The tab for dhry_1.c is displayed in the File Editor pane.

2. Ensure that the **Src** tab is selected, and display the line numbers in dhry_1.c by selecting **Edit → Editing Controls → Show Line Numbers**.

3. Set a trace start point at the start 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 in line 146. This line represents the start of a `for` loop. Select **Set/Toggle Trace Point...** from the context menu. A List Selection dialog is displayed.
   b. Select **Trace Start Point** from the List Selection dialog. 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 2-45.

![Figure 2-45 Setting a trace start point](image)
4. 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 201. (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 loop.) A context menu is displayed.
   b. Select **Set/Toggle Trace Point...** from the context menu. A List Selection dialog is displayed.
   c. Select **Trace End Point** from the List Selection dialog. An arrow is placed next to line 201 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 data or memory accesses that might be branched to between the points. For more details on these types of tracepoints, see *Setting simple tracepoints* on page 2-22.

5. Execute the image by selecting **Debug** → **Execution Control** → **Go (Start Execution)** from the Code window. The image is executed and a prompt is displayed in the Output pane at the bottom of the Code window.

6. 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.

7. To view the results of the trace capture, select **Tools** → **Analyzer/Trace Control** → **Display Profile View...** in the Code window. The Analysis window is opened, and the **Profile** tab is displayed, as shown in Figure 2-46 on page 2-103.
In the **Profile** tab, as shown in Figure 2-46, 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 *Column types* on page 2-60.

8. To view call-graph data for these results:
   a. Select **Parents of Function** from the **Profiling Data** menu in the Analysis window.
   b. Select **Children of Function** from the **Profiling Data** menu in the Analysis window.

The **Profile** tab displays parents and children of each function, as shown in Figure 2-47 on page 2-104.
Figure 2-47 Call-graph data displayed in the Profile tab

As well as 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 2-48 shows the data for a single function, Func_2.

The Exec% column shows that:

- 7.37% of the total execution time was spent in code of the function Func_2.
- 7.37% of the total execution time was spent in code of the function Func_2 when called from the parent main.
- 1.97% of the total execution time was spent in code of the function Func_1 when called as a child from Func_2.
- 9.47% 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:

- 18.82% of the total execution time was spent in calls to the function Func_2 and its children.
18.82% of the total execution time was spent in calls to the function Func_2 and its children when called from the parent main.

1.97% of the total execution time was spent in calls to the function Func_1 called as a child from Func_2.

9.47% 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 575 calls to the function Func_2.
- There were 575 calls to the function main to the function Func_2.
- There were 575 calls to the function Func_1 from the function Func_2.
- There were 575 calls to the function strcmp from the function Func_2.

### 2.6.4 Setting up a complex tracepoint

This example demonstrates how you can set up a complex tracepoint. It uses the dhrystone program described in *Capturing profiling information* on page 2-100. In this example, assume that you want to trigger trace in Proc_2, but only after 50 executions of the function Proc_1.

To perform this example:

1. Load the example image dhrystone.axf into the debugger. This file is located in the \Examples directory in your root installation. 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. The Trace on X after Y executed N times dialog is displayed, as shown in Figure 2-49.

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 textbox to display a menu of items saved in your personal history file. Select <Function List...> from this menu to display the Function List dialog. 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 textbox on the second line of the dialog, to specify that Proc_1 should be executed 50 times.

e. Select Instr Exec from the drop-down list in the last line of the dialog.

f. Click on the drop-down arrow to the right of the textbox to display a menu of items saved in your personal history file. Select <Function List...> from this menu to display the Function List dialog. 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 appears as shown in Figure 2-50.

![Figure 2-50 Completed complex tracepoint dialog](image)

Figure 2-50 Completed complex tracepoint dialog

g. Click OK to set the tracepoint as specified.

4. If you want to view the tracepoint, select View → Pane Views → Break/Tracepoints pane, or press Ctrl+5, to display the Break/Tracepoints pane, as shown in Figure 2-51.

![Figure 2-51 Break/Tracepoints pane with complex tracepoints set](image)

Figure 2-51 Break/Tracepoints pane with complex tracepoints set

5. View the Analysis window by selecting View → Analysis Window from the Code window. Ensure that Collect Trace Before Trigger is checked in the Trigger Mode submenu in the Edit menu.
6. Execute the image by selecting **Debug → Execution Control → Go (Start Execution)** from the Code window of RealView Debugger. The image is executed and a prompt is displayed in the Output pane at the bottom of the Code window.

7. In the Output pane (ensuring that the **StdIO** tab is selected), enter the number of runs through the benchmark you want RealView Debugger to perform, in this case at least 50.

8. The results of the trace capture are displayed in the Analysis window.
Tracing with RealView Debugger
Chapter 3
DSP Support

This chapter describes the Digital Signal Processor (DSP) support that is available in the RealView® Debugger. It contains the following sections:

- About DSPs and RealView Debugger DSP support on page 3-2
- Using the DSP on page 3-4.
3.1 About DSPs and RealView Debugger DSP support

RealView Debugger DSP includes support for:
- Oak and TeakLite processors from the DSP Group
- TeakDSP cores from the DSP Group
- Motorola M56621 DSP
- COFF image file format for the DSP Group Oak and TeakLite toolchain
- register definitions for Oak and TeakLite
- JTAG debug of DSP processors
- simulators for Oak and TeakLite processors.

RealView Debugger supports the Oak and TeakLite DSP engines produced by DSP Group. These are 16-bit processors designed to be integrated into custom or semi-custom silicon designs to provide extra signal processing performance.

You make a connection to a DSP Group processor, or to a simulated target, using RealView Debugger in exactly the same way as you make a connection to an ARM® processor. If the vehicle you are using supports the processor, it appears in the device list in the Connection Control window. See Using the DSP on page 3-4 for more details.

For more information on managing your connections, see the chapter describing connecting to targets in RealView Debugger v1.7 Target Configuration Guide.

3.1.1 Installing and using DSP support with RealView Debugger

Be aware of the following:
- Choose a Custom installation to install DSP support to ensure that the required JTAG files are available to enable connection using Multi-ICE® direct connect. You do not require a DSP license to connect to these targets.
- Currently, DSP support works only with remote targets containing an ARM7TDMI® core when connected through Multi-ICE direct connect.
- Choose a Custom installation to install DSP support if you want to use the DSP Group Oak DSP/TeakLite DSP, or the Motorola M56621 DSP. Do this to ensure that the required JTAG files are available. You must obtain a DSP license to connect to these targets.
- If you are using a DSP simulator, for example the MaxSim System-on-Chip (SoC) simulator from AXYS Design Automation, Inc., you must obtain a DSP license.
- You can use RealView ICE to connect to a target that incorporates DSP hardware with a suitable JTAG configuration. However, is is not currently possible to connect to DSP cores using RealView ICE.
3.1.2 Licensing and operating restrictions

RealView Debugger DSP support is separately licensed. You must obtain a license from your ARM distributor to use this feature.
3.2 Using the DSP

The DSP support in RealView Debugger is invoked by connecting the debugger to a suitable processor. This can be a simulated target or target hardware.

This section describes:
- Connecting to a simulator.
- Connecting to target hardware on page 3-7.

3.2.1 Connecting to a simulator

This example uses the MaxSim SoC simulator from AXYS Design Automation, Inc. to model multiprocessor debugging with an ARM920T™ core and a DSP (TeakLite), but the procedure for connecting is the same for any simulator.

If you are not licensed to use the multiprocessor facilities provided by RealView Debugger, you can still follow the simulator example but you can only make a single connection.

To access the DSP simulator with RealView Debugger:

1. Start the simulator and configure the processor model as required.
2. Start RealView Debugger to load the application files.
3. Select File → Connection → Connect to Target... to display the Connection Control window.
4. Expand the entry Server Connection Broker to display the available connections, shown in Figure 3-1.

![Figure 3-1 Simulator connections in the Connection Control window](image)
If you do not see the connections, you have not included DSP support in the RealView Debugger installation options. You must choose a Custom installation to install DSP support if you want to connect using a simulator.

5. Connect to the simulated targets by selecting the check box associated with each entry so that it is ticked, shown in Figure 3-2.

6. Click the Synch tab to set up processor synchronization if required, shown in Figure 3-3.

For more details on using the synchronization facilities in RealView Debugger, see Processor execution synchronization on page 5-45.

7. Select View → New Code Window to open a second Code window (RVDEBUG_1).

8. Click on the drop-down arrow on the Connection button and attach each window:
   • attach the default Code window (RVDEBUG) to the simulated ARM core
   • attach the second Code window (RVDEBUG_1) to the simulated DSP.
9. Click **File** → **Load Image...** to load an executable file to each of your targets, for example load:
   - ...\demo_DSP\ARM_Oak\ARM_iface\Debug\iface.axf to the ARM core
   - ...\demo_DSP\ARM_Oak\Oak_dtmf\Debug\dtmf.a to the DSP.
   If you install DSP support, examples images are supplied as part of the base product. By default, these are located in:
   *install_directory\RVD\Tools* ...

10. Click **Go** to start both processors.

11. See the simulation running by switching between the available windows on your desktop.

12. Debug your application in the usual way, for example, set breakpoints and step through your code, or change register entries and view the results in the simulation, shown in the example in Figure 3-4.

![Figure 3-4 Register view in simulator](image)

--- **Note** ---
If you end a debugging session, and close down RealView Debugger, this does not terminate RealView Connection Broker. You must shut down **RVBROKER** explicitly.
Failing to connect

If you do see a connection to a simulated target in the Connection Control window but cannot connect, check your licenses.

Use this FLEXlm command in a Windows command prompt or DOS box:

```
lmutil lmstat -a
```

**Note**
See Chapter 5 *Working with Multiple Target Connections* for information about using RealView Debugger with multiprocessor systems and windows attachment.

### 3.2.2 Connecting to target hardware

You connect to DSP target hardware in the same way as other hardware targets. However, you must use a suitable JTAG interface unit such as ARM RealView ICE or Multi-ICE.

**RealView ICE**

You can use RealView ICE to connect to a target that incorporates DSP hardware with a suitable JTAG configuration file. For example, suppose that your target contains an ARM core and a DSP core. You can use RealView ICE to connect to the ARM core if the configuration file for the SoC specifies that the DSP TAP controller is ignored, that is, by setting it to bypass. Doing this makes the DSP invisible to the debugger but enables you to connect to the ARM core.

For full details on how to configure a connection this way, see the chapter describing configuring custom connections in *RealView Debugger v1.7 Target Configuration Guide*.

**Multi-ICE direct connect**

You cannot use the normal ARM Multi-ICE configuration to connect to the DSP Group processors, because the Multi-ICE software does not support non-ARM architecture processors. Instead, you can use Multi-ICE direct connect.

Multi-ICE direct connect uses the Multi-ICE hardware and software within RealView Debugger to connect to target hardware that supports On-Chip Debugging (OCD). In this configuration, you require the Multi-ICE parallel port driver and the Multi-ICE interface hardware. However, the Multi-ICE Server must not be running.
To use Multi-ICE direct connect, use the ARM-ARM-PP connection in the Connection Control window. You must define the JTAG configuration file, for example using the Device JTAG-File Editor dialog, before you can connect. For full details on how to do this, see your Multi-ICE User Guide.
Chapter 4
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 4-2
- Using RealView Debugger RTOS extensions on page 4-7
- Connecting to the target and loading an image on page 4-22
- Associating threads with views on page 4-27
- Working with threads in the Process Control pane on page 4-34
- Using the Resource Viewer window on page 4-40
- Debugging your RTOS application on page 4-46
- Using CLI commands on page 4-58.
4.1 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.

In multiprocessor systems, specific processes and threads can be run on specific processors. For example:

- Processor 1 is dedicated to a specific task, for example, car engine timing. This is a single process with no threads.
- Processor 2 has multiple jobs, for example both displaying the fuel economy and processing radio-key messages. The developers implement these tasks as different processes, some of which have many threads.

### 4.1.1 Debugging an RTOS application with RealView Debugger

Debugging real-time systems presents a range of problems. This is especially true where the software being debugged interacts with physical hardware, because you normally cannot stop the hardware at the same time as the software. In some real-time systems, for example disk controllers, it might be impossible to stop the hardware.

**Note**

RealView Debugger can support a single RTOS connection or it can be used to debug multithreaded applications running on multiple processors.
Running and Halted System Debug

RealView Debugger supports different debugging modes, depending on the RTOS you are using:

Halted System Debug

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

However, HSD mode might not be suitable for real-time systems where stopping the debug target might damage your hardware, for example disk controllers.

Running System Debug

Running System Debug (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a Debug Agent (DA) that resides on the target and is typically considered to be part of the RTOS. The Debug Agent is scheduled along with other tasks in the system. See Debug Agent for details.

RSD mode is intrusive because it uses resources on your debug target and makes demands on the application you are debugging. However, this debugging mode provides extra functionality not available when using HSD, for example, RSD enables you to debug threads individually or in groups, where supported by your RTOS.

RealView Debugger enables you to switch seamlessly between RSD and HSD mode using GUI controls or CLI commands. For details see:

- Working with threads in the Process Control pane on page 4-34
- Using the Resource Viewer window on page 4-40
- Using CLI commands on page 4-58.

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 4-1 on page 4-5.
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.
4.2 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 Goto RTOS Awareness Downloads from the Code window Help menu for more information.

This section describes:

- Enabling RTOS support
- Creating a new RTOS-enabled connection on page 4-8
- Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file on page 4-12
- Configuring an RTOS-enabled connection without a vendor-supplied .bcd file on page 4-14
- Managing configuration settings on page 4-21.

4.2.1 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 \lib directory.
2. Copy the *.bcd file(s) into the \etc directory.

The Board/Chip definition file contains the information required to enable RTOS support in the debugger.
4.2.2 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 Debugger v1.7 Target Configuration Guide.

To set up the new connection:

1. Start RealView Debugger but do not connect to a target.

2. Select File → Connection → Connect to Target... to display the Connection Control window, shown in Figure 4-2.

   You can also click the Connection Control button, in the Connection group on the Actions toolbar, to display the Connection Control window quickly. If the window is hidden, click the button twice.

   ![Figure 4-2 Connection Control window](image)

Figure 4-2 shows the default connections set up after you first install RealView Debugger. The contents of this window depend on the autodetected targets available to you. If you have installed a Custom configuration your window looks different.
Note

If the 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 4-3.

![Figure 4-3 CONNECTION groups in the Connection Properties window](image)

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 4-4 on page 4-10.
Figure 4-4 Specifying a new CONNECTION group

Leave the type of the new entry unchanged as CONNECTION.

In the Group Name data field change the name from 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 *rv1_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 4-5.

**Figure 4-5 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 Debugger v1.7 Target Configuration 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 should 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. This is the opposite order to that used by Multi-ICE.
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 4-14.

### 4.2.3 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 4-5 on page 4-11.

2. Right-click on the new connection and select Connection Properties... from the context menu to display the Connection Properties window. Use this to configure your board file.

3. Expand the (*.bcd) Board/Chip Definitions entry, in the left pane, to see a full list of all .bcd files detected by RealView Debugger. This includes the vendor-supplied file copied earlier (see Enabling RTOS support on page 4-7).

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 4-6 on page 4-13.
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 4-20 for notes on working with multiple .bcd files.

6. Select File → Save and Close to save your changes to the board file and close the Connection Properties window.

7. Connect to your RTOS-enabled target and load an image, as described in Connecting to the target and loading an image on page 4-22.

   If you are accessing an RDI target for the first time, for example Multi-ICE®, it must be configured before it can be used.

   --- 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 4-21 for details.
There are descriptions of the general layout and controls of the RealView Debugger settings windows, including the Connection Properties window, in the RealView Debugger online help topic *Changing Settings*.

For full details on the tasks in this example, and how to configure RealView Debugger targets for first use, see the chapter describing configuring custom targets in *RealView Debugger v1.7 Target Configuration Guide*.

### 4.2.4 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-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 4-5 on page 4-11.

2. Right-click on the new connection and select *Connection Properties...* from the context menu to display the Connection Properties window. Use this to configure your board file.

3. Double-click on the `Advanced_Information` group, in the right pane, to expand the `Advanced_Information` block.

4. Double-click on the `Default` group, in the right pane, to see the RTOS and `ARM_config` groups.

5. Double-click on the RTOS group, in the right pane, to see the RTOS configuration settings, shown in the example in Figure 4-7 on page 4-15.
As shown in Figure 4-7, configure RTOS operation in your board file using the RTOS group:

**Vendor**
This three letter value identifies the RTOS plugin, that is the *.dll file supplied by your vendor.

**Load_when**
Defines when RealView Debugger loads the RTOS plugin:
- load the plugin on connection, with Load_when set to **connect**
- wait until an RTOS image is loaded, with Load_when set to **image_load**.

The RTOS features of the debugger are not enabled until the plugin is loaded and has found the RTOS on your target.

When the plugin is loaded, it immediately checks for the presence of the RTOS. If loaded, the plugin also checks when you load an image. This means that you might have to run the image startup code to enable RTOS features in the debugger.

**Base_address**
Defines a base address, overriding the default address used to locate the RTOS data structures. See your RTOS documentation for details.

**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 4-40 and Working with threads in the Process Control pane on page 4-34 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.

  Otherwise, contact is made with the Debug Agent but RSD is not fully operational. In this case, you can read memory while the target is running.

  *See OS marker in the Process tab on page 4-34 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 4-18 for more details on these settings.

Remember to select File → Save and Close to save your changes to the board file and close the Connection Properties window.

ARM configuration options

For ARM-based targets, you must also specify the ARM_config settings group in the Advanced Information block in your board file, shown in Figure 4-8.

![Figure 4-8 ARM_config group in the Connection Properties window](image)

For this group, configure the following settings:

1. Set Vector_catch to False.
   
   This is commonly used when debugging embedded systems to control how ARM exceptions are caught by the debugger. To implement RSD breakpoints, the Debug Agent must catch undefined exceptions. Ensure that you configure this setting as described.

2. Double-click on the Vectors group so that the contents are displayed in the right pane.
3. Define these settings as required by your application, for example set all options to False.

4. Define other settings as required by your application, for example expand the Semihosting group and set Enabled to False.

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 4-12.

---

**Connect and disconnect configuration options**

If you want to specify how RealView Debugger connects to (or disconnects from) a target processor, you can configure this in your board file. These definitions are contained in the `Advanced Information` block.

The configuration settings `Connect_mode` and `Disconnect_mode` are a special case when used to configure a debug target:

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

- If a (non-prompt) user-defined setting is specified in your board file and in any `.bcd` file linked to the connection, the board file setting takes priority.

- A blank entry in the top-level `Advanced Information` block ensures that any setting in a linked Board/Chip definition file is used instead. This might be important if you are using a vendor-supplied `.bcd` file to enable RTOS awareness (see *Configuring an RTOS-enabled connection to reference a vendor-supplied `.bcd` file* on page 4-12 for details).

To configure connect and disconnect behavior for the new connection:

1. Ensure that you can see the new connection in the Connection Control window, shown in Figure 4-5 on page 4-11.

2. Right-click on the new connection and select **Connection Properties...** from the context menu to display the Connection Properties window. Use this to configure your board file.
3. Double-click on the Advanced Information group, in the right pane, to expand the Advanced Information block.

4. Double-click on the Default group, in the right pane, to see the Connect_mode and Disconnect_mode settings, shown in the example in Figure 4-9.

As shown in Figure 4-9, configure connection behavior in your board file using the settings:

- Connect_mode
- Disconnect_mode.

Right-click on each setting to see the options available to you. These options are fixed and so might include options that are not supported by your target vehicle. If you specify such an option, the debugger prompts you to select an appropriate mode when you try to connect or disconnect.

For example, if you do not want to stop the target but still want to enable RSD mode, set Connect_mode to no_reset_and_no_stop from the context menu.

Remember to select File → Save and Close to save any changes to the board file and close the Connection Properties window.

Note

The connect (or disconnect) mode that is actually used depends on the target hardware, the target vehicle, and the associated interface software that manages the connection. If you are using 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 Debugger v1.7 Target Configuration 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 4-5 on page 4-11.
2. Right-click on the new connection and select **Connection Properties...** from the context menu to display the Connection Properties window.
3. Expand the (*.bcd) Board/Chip Definitions entry, in the left pane, to see a full list of all .bcd files detected by RealView Debugger.
4. Right-click on the entry BoardChip_name, in the right pane, and select the required group from the list, for example AP.
5. Select **File → Save and Close** to save your changes to the board file and close the Connection Properties window.
6. Connect to your RTOS-enabled target and load an image, as described in Connecting to the target and loading an image on page 4-22.

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 4-21 for more details.

- It is recommended that you do not edit any settings in a supplied .bcd file in case these change in a future release of RealView Debugger, or if they are updated by your RTOS vendor. If required, define custom files by creating new target descriptions as explained in the chapter describing configuring custom targets in RealView Debugger v1.7 Target Configuration Guide.

- Configuration settings defined as part of a project take precedence. Ensure that project settings do not conflict with target configuration settings, see RealView Debugger v1.7 Project Management User Guide for details.
4.2.5 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.

Where settings conflict, priority depends on the type of setting, whether it has changed from the default, and its location in the configuration hierarchy. When you reference multiple boards, RealView Debugger merges the settings to generate a combined configuration from all the matching groups. If the same setting is specified in more than one group, the specification in the group that is listed first in the CONNECTION is used.

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 File → Connection → 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.
4.3 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
- RTOS Exit Options
- Interrupts when loading an image
- Resetting OS state
- Loading from the command line

4.3.1 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 4-7.
- create a new RTOS connection as described in Creating a new RTOS-enabled connection on page 4-8.
- configure your RTOS-enabled connection as described in either:
  - Configuring an RTOS-enabled connection to reference a vendor-supplied .bcd file on page 4-12
  - Configuring an RTOS-enabled connection without a vendor-supplied .bcd file on page 4-14.

4.3.2 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 File → Load Image... to load the image.

If you have several executable files to load, use the ADDFILE and RELOAD commands.
4.3.3 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 but the application symbols should be 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 4-18.

If you have configured your connection to reference a vendor-supplied .bcd file that defines nonstop running, the connection options are defined by these settings and take precedence over any settings elsewhere. In this case, it is not necessary to specify the connection mode in your board file.

**Specifying connect mode**

When you connect to a running target, you can override any settings in your board file, or in any referenced vendor-supplied .bcd file, to specify the connection mode used. This option is also available when you disconnect from a target without stopping (see *Specifying disconnect mode* on page 4-24 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 File → 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 File → Connection → 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 Debugger v1.7 Target Configuration Guide for full details on connect, and disconnect, modes.

4.3.4 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 4-18 for details).

When you unload (or reload) an image, for example using the Process tab context menus, the Exit_Options setting decides how to disable RTOS awareness. If this is set to Prompt_on_Unload, the default setting, RealView Debugger displays a selection box to enable you to specify the exit conditions, shown in Figure 4-10 on page 4-25.
Figure 4-10 RTOS Exit Options selection box

This selection box offers:

**Do nothing**  Select this to maintain the current state.

**Terminate RSD**
Select this to disable RSD and end communication with the Debug Agent.

**Terminate HSD (and RSD)**
Select this to end communication with the Debug Agent and unload the RTOS plugin, that is the *.dll file.

Select the required state and then click **OK** to close the selection box.

If you click **Cancel**, this is the same as choosing **Do nothing → OK**.

### 4.3.5 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 **GO** command, an IRQ might occur associated with the previous image that could cause the current image to run incorrectly.

### 4.3.6 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.
4.3.7 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.

2. Provide the executable filename and any required arguments on the command line.

Some RTOS file loaders support the `target_name` parameter, which enables you to modify the file load actions. Refer to the documentation for your RTOS plugins for more information.
4.4 Associating threads with views

When HSD or RSD is fully operational, you can start to work with threads in the RealView Debugger Code window. This is described in the following sections:

- Attaching and unattaching windows
- The current thread on page 4-28
- Using the Thread button on page 4-28
- Working with the thread list on page 4-29
- Using the File menu on page 4-33.

You can also work with threads in the Process Control pane, see Working with threads in the Process Control pane on page 4-34 for details.

4.4.1 Attaching and unattaching windows

If you are licensed to use multiprocessor debugging mode, RealView Debugger Code windows can be attached to specific connections. Similarly, if you are working with an RTOS-enabled connection, you can attach Code windows to threads.

--- Note ---

Attaching windows to threads does not work in the same way as attaching windows to connections in multiprocessing mode, see Chapter 5 Working with Multiple Target Connections for details.

---

The windows attachment status is displayed in the Code window title bar, after the name of the target:

- [Unattached] Specifies that the Code window is not attached to a connection, that is a debug target board or a specified processor on a multiprocessor board. By default, unattached Code windows display details of the current thread, that is the thread that was most recently running on the target when the target stops.

- [Board] Specifies that the Code window is attached to a connection, that is a debug target board or a specified processor on a multiprocessor board. This window displays details of the current thread on that target, if available.

- <blank> If the title bar contains no attachment details then this window is attached to a specified thread, that is it is always associated with this thread.
**Note**

If you use View → New Code Window to create a new Code window, it inherits its attachment from the calling window.

When working with threads, you can change the attachment of your Code window using the thread list, see Working with the thread list on page 4-29 for details.

### 4.4.2 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.

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 Thread button, see Using the Thread 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 Debugger v1.7 Command Line Reference Guide for a full description of the THREAD command.

### 4.4.3 Using the Thread button

When HSD or RSD is fully operational, this enables the Thread button and drop-down arrow on the Actions toolbar in the Code window:

- Use the Thread button to view thread details and change the current thread, see Viewing thread details on page 4-29 for details.
- Use the **Thread** button drop-down to access the thread list where you can change your thread view and windows attachment, see *Working with the thread list* for details.

**Viewing thread details**

Use the **Thread** button to view each of the threads on the system in turn. Click the **Thread** 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 **Thread** 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 **Thread** 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 4-37 for details.

### 4.4.4 Working with the thread list

Click on the **Thread** 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 4-11 on page 4-30.

**Note**

In this release, the Debug Agent handles up to 64 threads. When the thread buffer is full, no threads can be displayed.
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 4-31 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 4-11, the fields shown (from left to right) for each thread are the:

- address of the thread control block
- name of the thread
- priority of the thread
- status of the thread (for example, ready, sleeping, or suspended event)
- thread suspend flags.

Click on a new thread, in the thread list, to change the code view so that it displays the registers, variables, and code for that thread:

- If you click on a new thread in an unattached Code window, it becomes attached to the thread automatically. The thread details appear in the Code window title bar and the Color Box changes color.
  
  If you change the thread view in this way, other unattached windows are not affected, that is they remain unattached and continue to show the current thread.

- If you click on a new thread in a Code window that is attached to a connection, it becomes attached to the thread automatically.

- If you click on a new thread in a Code window that is attached to a thread, it becomes attached to the specified thread.

In this release, the Debug Agent handles up to 64 threads. Where the thread list does not show the full details, use the thread selection box to see all the threads detected on the system (see *Using the thread selection box* on page 4-32 for details).
Current thread

As described in *The current thread* on page 4-28, RealView Debugger designates a thread to be the current thread when you are in RSD. In Figure 4-11 on page 4-30, 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 4-12 on page 4-32, the asterisk shows the current thread but any CLI commands operate on the attached thread, marked by a check mark.

See *Working with threads in the Process Control pane* on page 4-34 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 4-11 on page 4-30 shows two grayed threads:

- 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 **Thread** button drop-down arrow to display the thread list, shown in Figure 4-11 on page 4-30. 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 4-12.

![Figure 4-12 Example thread list in an attached window](image)

--- **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 **Thread** 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 4-37 for details.

--- **Note** ---

Attaching windows to threads does not work in the same way as attaching windows to connections in multiprocessing mode, see Chapter 5 *Working with Multiple Target Connections* 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 4-13 on page 4-33.
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.

### 4.4.5 Using the File menu

All the options available from the **Thread** button drop-down are also available from the Code window **File** menu.

Select **File → Thread** to see the thread list, shown in Figure 4-12 on page 4-32. From here you can view thread details and attach (or unattach) the Code window to a thread.
4.5 Working with threads 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 three 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:
- 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 RealView Debugger v1.7 User Guide 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 4-37.

### 4.5.1 OS marker in the Process tab

Select View → Pane Views → Process Control Pane to display the Process Control pane if it is not visible in your Code window. If HSD or RSD is enabled, the Process tab contains the OS marker, shown in Figure 4-14 on page 4-35.
Figure 4-14 RTOS OS marker in the Process Control pane (HSD)

Figure 4-14 shows a debug target where RealView Debugger has located the RTOS using the RTOS plugin. A suitable RTOS-enabled image has been loaded to the target. However, RealView Debugger has not detected that the RTOS has started.

In this state, neither the **Thread** tab nor the **Thread** button is available.

Figure 4-15 RTOS OS marker in the Process Control pane (RSD)

Figure 4-15 shows a running target where RealView Debugger where a suitable RTOS-enabled image has been loaded to the target and threads are being rescheduled.

In this state, the **Thread** tab is available and the **Thread** button is enabled on the Actions toolbar.
Table 4-1 describes OS marker status in the Process Control pane.

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT INITIALIZED</td>
<td>HSD or RSD is enabled but the triggering event that loads the plugin, the *.dll file, has not occurred.</td>
</tr>
<tr>
<td>HSD (PENDING)</td>
<td>The plugin has been loaded and RealView Debugger is waiting for it to determine that the RTOS has been found.</td>
</tr>
<tr>
<td>HSD (PENDING OS)</td>
<td>The plugin has found the RTOS but the process is not yet in a state where threads are being rescheduled.</td>
</tr>
<tr>
<td>HSD</td>
<td>HSD is fully operational.</td>
</tr>
<tr>
<td>HSD (RUNNING)</td>
<td>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.</td>
</tr>
<tr>
<td>RSD (PENDING DA)</td>
<td>RealView Debugger is waiting for notification that a Debug Agent has been found.</td>
</tr>
<tr>
<td>RSD</td>
<td>RSD is fully operational.</td>
</tr>
<tr>
<td>HSD (RSD STOPPED)</td>
<td>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.</td>
</tr>
<tr>
<td>HSD (RSD INACTIVE)</td>
<td>RSD was operational or pending but is now disabled. This occurs only by a user request. HSD is now available.</td>
</tr>
<tr>
<td>HSD (RSD DEAD)</td>
<td>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.</td>
</tr>
<tr>
<td>RSD DEAD</td>
<td>RSD was operational but is now disabled. The debug target is still running. HSD is not available.</td>
</tr>
<tr>
<td>RSD (PENDING SYMBOLS)</td>
<td>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.</td>
</tr>
</tbody>
</table>
The OS marker is usually shown in black text in the **Process** tab. Where the marker is shown in red, there has been an error or RTOS support is not ready, shown in Figure 4-14 on page 4-35.

**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 Working with RTOS resources on page 4-41.

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 4-14 for details.

**Stop Target Processor**

Stops the target processor and suspends RSD. Depending on your configuration settings, click the **Go** button to start execution and restart RSD.

**Properties** Select this option to see details about the Debug Agent. This includes the status of the RSD module, settings as specified in your board file (or .bcd file where available), and RSD breakpoints. For an example, see More on breakpoints on page 4-47.

### 4.5.2 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. This pane is available in both HSD and RSD mode. In RSD, however, the **Thread** pane shows a snapshot of the last known state of the system.

Select **View** → **Pane Views** → **Process Control Pane** 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, select **View** → **Pane Views** → **Threads**.

________

Expand the tree to see each configured thread and the associated summary information, shown in Figure 4-16 on page 4-38.
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 4-16 shows that the Code window is attached to thread 3.

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.
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 4-28).

**Attach Window to**
Attach the Code window to this thread (see *Attaching windows to threads* on page 4-31).

**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 4-4 for details.
4.6 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.

Select View → Resource Viewer Window to display this window, shown in Figure 4-17.

![Resource Viewer Window](image.png)

**Figure 4-17 Resource Viewer showing thread details**

If you are in HSD mode, you must stop execution to view your RTOS resources.

The Resource Viewer window title bar reflects the title bar of the calling Code window. In this example, the Code window is unattached and so is showing the current thread.

This section describes the Resource Viewer window:

- **Changing connections**
- **Working with RTOS resources** on page 4-41
- **Using Action menus** on page 4-44.

### 4.6.1 Changing connections

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.
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.

In multiprocessor debugging mode, the Conn tab shows all active connections and an asterisk (*) indicates the current connection. See the Chapter 5 Working with Multiple Target Connections for full details.

### 4.6.2 Working with RTOS resources

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. In multiprocessor debugging mode, this applies across all connections.

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

Auto Update on Timer...
Not available in this release.

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 4-14 for details.
If RSD is enabled, select Disable RSD to shutdown the Debug Agent cleanly. You can re-enable RSD after it has been disabled.
If you select Disable RSD, this disables all previously set RSD breakpoints. However, all HSD breakpoints are maintained.
If you select Enable RSD, system breakpoints are enabled but you have to enable thread and process breakpoints yourself.

Stop Target Processor
Stops the target processor, suspends RSD, and switches to HSD. Depending on your configuration settings, click the Go button to start execution and restart RSD.

Properties
Select this option to see details about the Debug Agent. This includes the status of the RSD module, RTOS-specific settings as defined in your board file (or .bcd file where available), RSD breakpoints, and symbols.
**Note**

The RSD menu includes the same options as the OS marker context menu in the Process Control pane. See *OS marker in the Process tab* on page 4-34 for details.

**Resources toolbar**

This toolbar contains three controls:

**Connection button**

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

**Note**

The Connection button and drop-down do not work in the same way as the Thread button and drop-down, see Chapter 5 *Working with Multiple Target Connections* for full details on using this button.

**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 threads in the Process Control pane* on page 4-34 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 4-17 on page 4-40). Click on the thrd tab to see the thread list available from the Thread 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 4-11 on page 4-30 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 4-17 on page 4-40).

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 4-17 on page 4-40).

For more information about the meaning of the tabs and information displayed in the Resource Viewer window, refer to 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.

4.6.3 Using Action 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 menu.

The available actions, and the associated parameters, depend on the Debug Agent. In the example shown in Figure 4-17 on page 4-40, select the threads view and right-click on a specific thread to see the associated Action menu options. In this case, the Debug Agent offers the possible actions:
- delete
- suspend
- resume.

The Action 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 menu that requires parameters, a prompt is displayed, shown in the example in Figure 4-18 on page 4-45.
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 4-31 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.

Be aware of the following:

- The thread status, as shown in the Resource Viewer window, is fully integrated with the thread list as shown in the Code window.

- Using actions to suspend, or resume, threads is not fully integrated with RealView Debugger:
  - If you issue an action command to suspend a specified thread, 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 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.

See *RTOS action commands* on page 4-64 for details on using CLI commands to carry out actions on RTOS objects. For more information about the meaning of options in the **Action** menu, refer to the user manual for your RTOS.
4.7 Debugging your RTOS application

For full details on how to debug your applications using RealView Debugger, see RealView Debugger v1.7 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 4-48
- Using the Set Address/Data Break/Tracepoint dialog box on page 4-51
- Using the Break/Tracepoints pane on page 4-52
- Stepping threads on page 4-53
- Manipulating registers and variables on page 4-55
- Updating your debug view on page 4-56.

4.7.1 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 → Complex Breakpoints → 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 4-34 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 Debugger v1.7 User Guide for a full description of HSD breakpoints in RealView Debugger.
- the chapter describing memory mapping in RealView Debugger v1.7 User Guide for a full description of memory types and access rules.

### 4.7.2 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. Right-click in the gray area to the left of a line to see the context menu, shown in Figure 4-19.

![Figure 4-19 Setting a thread breakpoint quickly](image)

---

**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 (double click)** 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, shown in Figure 4-20.

**Figure 4-20 Changing the break trigger group**

--- 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 4-58 for details).

4.7.3 Using the Set Address/Data Break/Tracepoint dialog box

Use the Set Address/Data Break/Tracepoint dialog box to set breakpoints:

1. Right-click in the gray area to the left of a line to see the context menu, shown in Figure 4-19 on page 4-49.

2. Select Set Break... to display the Set Address/Data Break/Tracepoint dialog box, shown in part in Figure 4-21.

Ensure that you select Set Break... If you select Set Break (double click), the Set Address/Data Break/Tracepoint dialog box dialog box is not displayed and a system breakpoint is set automatically.

![Figure 4-21 RTOS Breakpoint Class selector](image-url)
Note

See the chapter describing working with breakpoints in RealView Debugger v1.7 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.

4.7.4 Using the Break/Tracepoints pane

Select View → Pane Views → Break/Tracepoints Pane from the Code window main menu to display the Break/Tracepoints pane in the usual way, shown in Figure 4-22.

Figure 4-22 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 4-48 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 4-22 on page 4-52 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 4-23 shows three breakpoints are set in HSD mode. Here, the extra breakpoint details (available in RSD) are not included 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 4-22 on page 4-52 and Figure 4-23 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 Debugger v1.7 User Guide for a full description of the Break/Tracepoints pane.

4.7.5 Stepping threads

Use the Execution group, from the Actions toolbar, to control program execution, for example starting and stopping execution, and stepping through a multithreaded application. These options are also available from the Execution Control submenu of 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 4-24 represents three threads at different stages of execution.

Figure 4-24 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.

### 4.7.6 Manipulating registers and variables

Select View → Pane Views → 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 Debugger v1.7 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 in order 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. In order to ensure that such modifications operate correctly, you must either:

- in RSD mode, modify the variable and then, if at the point you have stopped the relevant thread any thread has a cached copy of the variable, modify the copy as well
- in HSD mode, modify the variable and then, if at the point you have stopped the processor any thread has a cached copy of the variable, modify the copy as well
- in RSD or HSD, declare the variable to be volatile and recompile the program.

4.7.7 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 below.

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 Debugger v1.7 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.
4.8 Using CLI commands

You can use CLI commands when debugging your RTOS image:

- **OSCTRL**
- **BREAKINSTRUCTION** on page 4-59
- **HALT** on page 4-61
- **STOP** on page 4-62
- **RTOS resource commands** on page 4-63
- **RTOS action commands** on page 4-64
- **Getting more help** on page 4-65.

4.8.1 OSCTRL

The OSCTRL command controls OS Awareness.

**Syntax**

`OSCTRL ,qualifier [=value]`

where:

- **qualifier** Specifies the action.
- **value** Specifies a file where filters are saved.
  
  Not available in this release.

**Examples**

The following examples show how to use OSCTRL:

- `osctrl,enable_rsd`
  
  Enable RSD.

- `osctrl,disable_rsd`
  
  Disable RSD.

- `osctrl,properties_rsd`
  
  Report the current RSD properties in the Output pane, for example the status of the RSD module, settings as specified in your board file (or .bcd file where available), and RSD breakpoints.
4.8.2 BREAKINSTRUCTION

The BREAKINSTRUCTION command sets a breakpoint at the specified location.

Syntax

BREAKINSTRUCTION [{,qualifier...}] [expression] [{=threads,...}] [{macro-call}]

where:

- **qualifier** is an ordered list of zero or more qualifiers to identify the type of breakpoint. The possible RTOS qualifiers are `rtos:hsd`, `rtos:system`, `rtos:thread`.
  - If you do not specify a qualifier, `rtos:hsd` is assumed.
  - The `rtos:process` qualifier is not available in this release.
- **expression** specifies the address at which the breakpoint is placed. For an unqualified breakpoint, this is the address at which program execution is stopped.
- **threads** is the list of threads that make up the break trigger group.
  - Only available for thread breakpoints in this release.
- **macro-call** specifies a macro and any parameters it requires.
  - The macro runs when the breakpoint is triggered and before the instruction at the breakpoint is executed. The macro is treated as being specified last in the qualifier list.

Examples

The following examples show how to use BREAKINSTRUCTION:

- `breakinструкtion, rtos:hsd \DEMO\#201`
  - Set an HSD breakpoint at line 201 in demo.c.
- `bi,rtos:system \DEMO\#154`
  - Set a system breakpoint at line 154 in demo.c.
- `bi,rtos:thread \DEMO\#154 = 0x39d8, 0x3a68`
  - Set a thread breakpoint using a break trigger group consisting of two threads, defined by the TCB addresses.
bi,rtos:thread \\DEMO\\#180 = thread_2, thread_6, thread_8

Set a thread breakpoint using a break trigger group consisting of three threads, defined by the thread names.

bi,modify:2,rtos:system

Modify a thread-specific breakpoint to a system breakpoint.

bi,modify:3,rtos:thread = 0x1395c, 0x13bac

Modify a thread breakpoint to specify a different break trigger group, shown in Figure 4-25.

![Figure 4-25 Changing the break trigger group](image)
4.8.3 HALT

The HALT command stops target program execution. In RSD mode, this stops the current thread.

Syntax

HALT

Examples

The following example shows how to use HALT:

halt      Stops the current thread.
4.8.4 STOP

The STOP command stops the target processor or a specified thread.
The STOP command is independent of the windows attachment or the current thread.

--- Note ---
You can only use the STOP command to stop threads in RSD. This is accomplished by the Debug Agent using the associated OS service.

Syntax

STOP [=value]

where:

value Identifies the thread.

Examples

The following examples show how to use STOP:

stop Stops the processor.

stop = thread_4

    Stops the named thread.

stop = 0x39d8

    Stops the thread specified by the TCB address.
4.8.5 RTOS resource commands

The dos_<resource-list> command displays an RTOS resource list or shows details of one element in that list.

Syntax

dos_<resource-list>,qualifier [=value]

where:

qualifier  Specifies what to display, that is all or detail.
value  Is the identifier, that is a control block (specified by TCB address) or resource name.

Examples

The following examples show how to use dos_<resource-list>:

dos_thread_list,all
    Display all resources in the Output pane. This displays the thread details, shown in Figure 4-17 on page 4-40.

dos_thread_list,detail = 0x39d8

dos_thread_list = 0x39d8
    Display details about the thread, specified by TCB address, in the Output pane. This displays the information as shown in the Details area of the Resource Viewer window, shown in Figure 4-17 on page 4-40.

dos_thread_list,detail = thread_4
    Display details about the named thread in the Output pane. This displays the information as shown in the Details area of the Resource Viewer window, shown in Figure 4-17 on page 4-40.

dos_timer_list,detail = 0x39d8
    Display details about the specified timer in the Output pane. This displays the information as shown in the Details area of the Resource Viewer window.

——— Note ————

RTOS resource CLI commands of the form $resource_LIST=expression are not available in this release. They have been replaced by dos_<resource-list> commands.
4.8.6 RTOS action commands

The `aos_<resource-list>` command performs an action on an object chosen from the RTOS resource list.

Syntax

`aos_<resource-list> ,qualifier [=value]`

where:

- **qualifier** Specifies the action, that is `<action-name>[<parameter>]`.
  - **action-name** The action to perform, for example to set a timer.
  - **parameter** A value to use in the action, for example a new timer value.
- **value** Is the identifier, that is a control block (specified by TCB address) or resource name.

**Note**
The actions that you can specify, and the associated parameters, depend on the Debug Agent. Therefore, some actions, or parameters, might not be available.

**Examples**
The following examples show how to use `aos_<resource-list>`:

- `aos_thread_list,suspend = 0x39d8`
  
  Suspends the thread, specified by TCB address.

- `aos_timer_list,set:100 = 0x15260`
  
  Set the specified timer to 100.

- `aos_timer_list,deactivate=timer_1`
  
  Deactivate the specified timer.
4.8.7 Getting more help

When you are using the CLI:

- To see a full list of RealView Debugger commands enter either:
  
  `dhelp=all`
  `dcommands=all`

- To see details of a specific command enter, for example:
  
  `dhelp,full osctrl`
  `dhelp,full break`

- To see a full list of RTOS resource commands enter, for example:
  
  `dhelp,status =all`

See *RealView Debugger v1.7 Command Line Reference Guide* for a full description of all CLI commands, including `BREAKINSTRUCTION` and the RTOS commands described in this section.
Chapter 5
Working with Multiple Target Connections

This chapter describes in detail the features of RealView® Debugger that enable you to make more than one connection at a time. This helps debug multiprocessor applications and compare the behavior of different targets, for example two ARM® processors.

This chapter contains the following sections:

- Overview of multiple target connections in RealView Debugger on page 5-2
- The RealView Debugger multiprocessor architecture on page 5-3
- Managing multiple targets on page 5-13
- Display coherency on page 5-37
- Processor execution synchronization on page 5-45.
5.1 Overview of multiple target connections in RealView Debugger

This chapter introduces the concepts of multiprocessor debugging, and how it is implemented with RealView Debugger.

--- Note ---

The multiprocessor facilities provided by RealView Debugger are separately licensed. You must obtain a license to use these facilities.

See The RealView Debugger multiprocessor architecture on page 5-3 for a description of the overall approach that RealView Debugger takes to managing multiple target connections. This section briefly describes how connections are set up and how you can select the connection that is relevant for a specific purpose.

Managing multiple targets on page 5-13 describes how to manage multiple debug targets using the RealView Debugger interface.

Display coherency on page 5-37 describes how the issues of coherency, mostly memory coherency, affect you, and explains the measures you can take to avoid problems resulting from an incoherent view of the target. This section includes a worked example, setting up a board file for a three processor system including shared and local memory.

Processor execution synchronization on page 5-45 describes how you can synchronize debugger start and stop requests across processors in your debug target system. It includes an explanation of the different ways to synchronize processors and how to include or exclude some processors from the synchronized group.
5.2 The RealView Debugger multiprocessor architecture

RealView Debugger supports debugging multiple processors on one target system in different ways, for example:

- with multiple simulator connections using RealView ARMulator® ISS (RVISS)
- using a single target hardware connection (for example, a JTAG scan chain)
- using multiple connections (for example, a JTAG scan chain and a debug monitor).

RealView Debugger supports this by separating the target connection from your view of that connection. This enables you to decide which connection to examine without having to disconnect and reconnect the debugger.

This section describes the RealView Debugger multiprocessor architecture. It contains the following sections:

- Connecting to a single target
- Connecting to two targets on page 5-6
- Connecting to multiple targets on page 5-9
- Using target debug interfaces on page 5-11.

5.2.1 Connecting to a single target

Figure 5-1 on page 5-4 shows the relationship between a single processor (in this instance, an ARM processor), the debugger, and a single Code window.

Note

The example shown in Figure 5-1 on page 5-4 is used to show the relationship between the different components that make up the RealView Debugger multiprocessor architecture. This is not intended as a real-world example since, in general, debug targets consist of one ARM processor, two ARM processors, or an ARM and a DSP.
Figure 5-1 shows the relationship of these components:

**Code window**
The Code window is the starting point for all your debugging tasks and gives you access to features in the product.
The Code window is part of the RealView Debugger GUI and provides a user interface to the Application Programming Interface (API) of the debugger core. It uses the connection state information held in the debugger core to display code views, give access to other windows, handle user commands, and display debugger messages.
Debugger core
Satisfies requests from the code view by acting on the target debug interface. This component carries out API requests, for example to load a program to the target, translating them into sequences of operations on target memory and registers.

Connection state information
You access connection state information through the Code window. It describes how the debugger connects to the debug target, any information required to use that connection, and what kind of processor your target is using. It might also include cached copies of processor registers or memory.

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

RealView Debugger requires a suitable interface to control the processor, for example a JTAG interface such as ARM RealView ICE, connected across your network. You can also use a parallel port and JTAG interface hardware such as Multi-ICE®.

Target
The hardware and software system that you are creating or debugging.

The configuration shown in Figure 5-1 on page 5-4 describes the state RealView Debugger is in when you first connect to a target. This applies when you are working in single processor debugging mode, using the default Code window. In this example, you are using a RealView ICE interface unit to connect to a single ARM processor on your target board.

There is a distinction between the Code window, displaying target information, and the target connection itself:

- The host code that controls the Code window uses generic debugger operations, for example, to retrieve register contents for display in the Register pane. The host code can also change the target state, for example, to write to a variable in response to your request using the Watch pane.

The Code window maintains a reference to the current connection in the debugger core, represented on the figure by a line with a dot at one end. Change this using the windows attachment options, described in Using the Connection menu on page 5-20. If the connection reference is changed, the Code window refreshes each element of the display using the new connection, and so displays the state of the new target.
• The host code called by the Code window maintains data structures representing each connection. The data structures describe the processor, its current state, and the nature of the connection to the target, and enable the code to action the Code window requests.

Note
This also applies to other windows that you can call from the Code window, that is the Resource Viewer window and the Analysis window.

This distinction enables RealView Debugger to maintain a connection without requiring a window to display it, and to maintain more than one connection with only one window. The following example demonstrates this.

5.2.2 Connecting to two targets

Figure 5-1 on page 5-4 shows your first connection to the ARM processor on the debug target board.

Let us suppose that you now want to make a second connection to the DSP. RealView Debugger creates a new connection and new connection state information describing the connection. This configuration is shown in Figure 5-2 on page 5-7.

Note
The example shown in Figure 5-2 on page 5-7 is used to show the relationship between the different components that make up the RealView Debugger multiprocessor architecture. This is not intended as a real-world example since, in general, debug targets consist of one ARM processor, two ARM processors, or an ARM and a DSP.
If you compare Figure 5-1 on page 5-4 with Figure 5-2, you can see there is a new circle in the Debugger Core, representing the new connection state information. RealView Debugger has not deleted the previous connection to the ARM processor. This means that, although the Code window link (the thick line with a dot) is now referencing the new connection (the default behavior), the previous ARM connection is still available.

**Current connection**

The term *current connection* is used to denote the selection mechanism that the debugger uses to decide what to display and to provide the scope of CLI commands. The current connection is usually the last connection that you made.
Connections are queued in the order in which they were made. As a new connection is added to the list of available connections, it becomes the current connection. If you disconnect the current connection, the next available connection in the list automatically becomes current.

In Figure 5-1 on page 5-4, the ARM connection is the current connection. If you then connect to a second connection, the DSP, shown in Figure 5-2 on page 5-7, this becomes the current connection.

You can change the current connection yourself so that the Code window displays a different view of your target. See Changing the current connection on page 5-22 for details on how to do this.

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

Windows attachment

With a connection established, attachment refers to whether a window is tied to a particular processor. If a window is:

- **Attached** It only displays information of the attached connection.
- **Unattached** It only displays information of the current connection.

**Note**

If you are not licensed to work in multiprocessor debugging mode, all Code windows are unattached by default. However, this does not appear in the title bar.

Attachment enables you to use Code windows in a way that suits your working style, for example you might use a single unattached Code window and then cycle through the active connections displaying each current connection in turn. Alternatively, you might create multiple Code windows and attach each to a specified connection.

See Attaching windows to multiple connections on page 5-23 for details on configuring windows attachment.

Using a target debug interface

In the example shown in Figure 5-2 on page 5-7, you are now using a Multi-ICE interface unit to connect to both the ARM processor and the DSP on your target board. This is necessary because, at this stage of development, it is not possible to connect to DSP cores using RealView ICE.
You do not require multiple interface units to make connections to multiple processors in this way. However, you could use a RealView ICE unit to connect to the ARM processor and a simulator to connect to a DSP during your applications development stage or where real DSP hardware is not available.

See Using target debug interfaces on page 5-11 for more details.

5.2.3 Connecting to multiple targets

Following on from Figure 5-2 on page 5-7, Figure 5-3 on page 5-10 shows the state of the debugger after you make more connections and create a second Code window.

--- Note ---

The example shown in Figure 5-3 on page 5-10 is used to show the relationship between the different components that make up the RealView Debugger multiprocessor architecture. This is not intended as a real-world example since, in general, debug targets consist of one ARM processor, two ARM processors, or an ARM and a DSP.
Figure 5-3 Creating multiple connections and views on the target

Figure 5-3 shows:

- a connection to an ARM processor
- a connection to an ARM processor that is running a multithreaded application using an RTOS
- a new Code window and a new connection displaying the state of the DSP processor on the target.
If you compare Figure 5-3 on page 5-10 with Figure 5-2 on page 5-7, you can see there is a new circle in the Debugger Core, representing the new connection state information. The Code windows have been attached to specific connections so that they display details about only that connection.

For more information see Managing multiple targets on page 5-13.

Connecting to RTOS-enabled targets

Figure 5-3 on page 5-10 describes how the RealView Debugger multiprocessor architecture supports connections to RTOS-enabled targets.

If you connect to an RTOS-enabled target, and RealView Debugger detects the RTOS, the processor itself becomes invisible to the debugger. Instead, the Code window shows details of the current thread running on the processor, shown in Figure 5-3 on page 5-10.

Like the current connection:

- you can change the current thread in a Code window by selecting a new thread from the thread list
- a single Code window always displays information relating to the current thread unless you attach the window to another thread.

Note

RealView Debugger can support a single RTOS connection or it can be used to debug multithreaded applications running on multiple processors.

For more information on working with RTOS-enabled targets see Chapter 4 RTOS Support.

5.2.4 Using target debug interfaces

In these descriptions, the target debug interface unit has been largely ignored. However, as shown in Figure 5-2 on page 5-7 and Figure 5-3 on page 5-10, the way in which a connection is made is important, not least because of the limitations it can impose on you. This is because each debug target interface is capable of supporting different numbers of connections and different kinds of connections. Similarly, the type of processor you can connect to varies depending on the underlying debug target interface. Because of these factors, you might use multiple target debug interfaces to a single target board.
RealView Debugger supports multiple target debug interfaces by separating the target connection from your view of that connection. However, the behavior of RealView Debugger does not change, whether there is a single target connection or many connections.

Note
Remember, limitations inherent in these different interfaces might have an impact on the way the debugger operates. For example, the speed of download might vary between interfaces or some features might not be available on some interfaces.
5.3 Managing multiple targets

RealView Debugger provides support for both multiprocessor systems that have all processors on the same JTAG scan path, and for systems that mix JTAG and other forms of debug access. RealView Debugger enables you to examine and control processes running on several processors, and to view variables and registers through the user interface.

This section describes:
- The Connection Control window
- Viewing connection details on page 5-17
- Using the Connection menu on page 5-20
- Disconnecting from targets on page 5-25
- Working with projects on page 5-29
- Using RealView ICE with multiple targets on page 5-34
- Using Multi-ICE with multiple targets on page 5-35.

5.3.1 The Connection Control window

Like in single processor mode, the first step in a multiprocessor debugging session is to make one or more connections to your debug targets. With connections established, you can load images ready for debugging.

When you are working with multiprocessor debug systems, the Connection Control window provides the main window for making connections, managing those connections, and synchronizing processing operations.

If the Connection Control window is closed, or hidden by other windows, display it by selecting File → Connection → Connect to Target.... It is shown in Figure 5-4 on page 5-14.
Working with Multiple Target Connections

Figure 5-4 Connection Control window in multiprocessor mode

In multiprocessor mode, the Connection Control window shows two tabs:

**Connect**  
Like single processor debugging mode, this shows all the targets available to you as specified in your board file, for example rvdebug.brd, and associated target configuration files.

Select **File → Connection → Connect to Target...**, from the Code window, to display the Connection Control window. This opens the window with the **Connect** tab visible.

**Synch**  
Use this tab to synchronize (or unsynchronize) processors during your debugging session, and to view synchronization options for different targets.

Select **File → Connection → Synchronization Control...**, from the Code window, to display the Connection Control window. This opens the window with the **Synch** tab visible.

Expand the entries in the Name column to see the available connections, shown in Figure 5-4.

Establishing connections

In single processor debugging mode, you make your first connection to a debug target as described in *RealView Debugger v1.7 Essentials Guide*, for example by double-clicking on the required Name or Description.
To make further connections in multiprocessor mode, display the Connection Control window and repeat these steps for the required connection entries. If you do this with a single processor version of RealView Debugger, you get a dialog reporting:

You cannot connect to this core, as a license to access multiple processor cores could not be checked out.

With a connection established you can load an image. Select File → Load Image... to display the Load File to Target dialog box where you can locate the required image and specify the way in which it is loaded.

Making a second connection adds a new entry to the list of active connections and your new connection automatically becomes the current connection. Each time you make a new connection, or terminate a connection, the unattached Code window title bar is updated to show the connection details.

See Viewing connection details on page 5-17 for details on the contents of the title bar.

You can use the Connection Control window to see all the available connections during a debugging session, and to configure new target connections. See the chapter describing connecting to targets in RealView Debugger v1.7 Target Configuration Guide for more details on using this window.

Setting connect mode

You can control the way RealView Debugger connects to a processor. For example, you might want to connect to a target that is already running an application from a previous session. This is useful when debugging multiprocessor systems or multithreaded applications.

See the chapter describing connecting to targets in RealView Debugger v1.7 Target Configuration Guide for full details on using the Connection menu from the Connection Control window and specifying the state of a processor when you connect.

Changing connections

With two connections established, use the Connection button on the Actions toolbar to switch to a different target. Click on the Connection drop-down arrow to display the Connection menu to select the particular connection you want to view, shown in Figure 5-5 on page 5-16.
All the options available from the **Connection** button are also available from the Code window **File** menu.

Select **File → Connection** to see the list of active connections, shown in Figure 5-5. From here you can view each available connection, make a new connection the current connection, and attach (or unattach) the Code window to a connection.

An asterisk * beside the name of the connection in the connections list indicates the current connection. This is usually the last connection established. Unattached windows always display the current connection.

See *Using the Connection menu* on page 5-20 for details on using this menu.

If you want to return to displaying the state of the ARM processor in the Code window, you can do so by selecting the ARM connection in the Code window **Connection** drop-down list or by clicking on the **Connection** button to cycle through the connections in turn.

--- **Note** ---

This behavior changes if the Code window is attached, see *Windows attachment* on page 5-21 for details.

You can also make a connection to a processor running an RTOS on your debug target board. Figure 5-2 on page 5-7 also shows a CPU running an RTOS supporting four threads, represented by four small circles. This capability is described in more detail in Chapter 4 *RTOS Support*.

You can make further connections to your debug target in a similar way, building up a group of connection states in the debugger core. With these connections established, you can switch freely between them using a single Code window, or you can create more than one Code window, enabling you to view more than one connection on screen at once.

To create a new Code window, select **View → New Code Window** from the main menu in an existing Code window. This creates a new window referencing the same connection as the calling Code window.
Logging commands and output

As you establish each new connection, the **Cmd** tab of the Output pane keeps a record of all the commands submitted during the connection process and any messages returned by RealView Debugger, shown in Figure 5-6.

![Figure 5-6 Connection details in the Output pane](image)

### 5.3.2 Viewing connection details

As you establish new connections, view your connection details in:

- *The Code window title bar*

**The Code window title bar**

The Code window title bar gives details of the connection and any processes running on your debug target. If you connect to a target and load an image, your title bar looks like the one shown in Figure 5-7.

![Figure 5-7 Connection information in the Code window title bar](image)

This shows:

- **RVDEBUG** Identifies the Code window. This changes as you open new windows, for example RVDEBUG_1, or RVDEBUG_2.
(dhrystone) The active project. This might be an open user-defined project or the auto-project associated with a loaded image.

In RealView Debugger, a project can be associated with a connection, that is it is bound to that connection, shown in Figure 5-7 on page 5-17. See Working with projects on page 5-29 for details on project binding.

@SimARM... The connection, including the target vehicle and the connection number.

[Unattached] The attachment of the window to a specified connection.

If you are working in multiprocessor mode, the default Code window is unattached. You can attach a Code window to a specified connection, shown by [Board]. See Using the Connection menu on page 5-20 for details.

If you are working in multiprocessor mode and the title bar contains no attachment details then this window is attached to the current thread.

If you are working in single processor debugging mode, the option to attach windows to your connection is not available.

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

The Resource Viewer window

You can see more details about your connections using the Resource Viewer window:

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

2. If not already visible, click on the Conn tab to display the connections tab, shown in Figure 5-8 on page 5-19.
Working with Multiple Target Connections

The Resource Viewer only has multiple tabs if this is required. For example, connecting to an RTOS application with the RTOS extension enabled creates additional tabs to display RTOS resources.

3. Select one of the connections in the top pane, the Resources list.

4. Select View → Display Details from the menu bar to see details about the chosen connection. This information appears in the lower pane, the Details area.

You can display connection details in one step by double-clicking on the chosen connection. This immediately displays the details about that item.

The Resource Viewer window title bar reflects the title bar of the calling Code window when it first opens. If you are working with multiple Code windows, the Resource Viewer is updated to be consistent with the most recent change you make to the debugging environment. This means that the title bar and the Color Box change as you open (or close) Code windows, make new connections, change the current connection, or disconnect from targets.

If you change your current connection using the Connection button, the top pane in the Resource Viewer (the Resources list) is updated. The new current connection is marked by an asterisk. The current connection and the connection that was previously the current connection are colored blue to show that they have changed.

If you keep the Resource Viewer window open and make another connection to a debug target, the Resources list is refreshed. The new connection is the current connection and this is marked with an asterisk.

Figure 5-8 Multiple connections in the Resource Viewer window

For more information, select Help from Menu.
Note

If you change the attachment of the calling window after the Resource Viewer window opens, this change is not reflected in the title bar. This means that, if you are working with several Resource Viewer windows at the same time, the title bars might not accurately reflect the status of Code windows. Close and reopen the Resource Viewer to update the whole window.

5.3.3 Using the Connection menu

In multiprocessor mode the Connection button is added to the Code window toolbar and is enabled when at least one connection is made to a debug target. This button, and its associated menu, enables you to change the current connection and to manage windows attachment during your debugging sessions.

In the Code window, click on the drop-down arrow on the Connection button, to display the Connection menu, shown in Figure 5-9.

![Figure 5-9 Connection menu](image)

This menu contains two components:

**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, as in Figure 5-9. The Code window title bar reflects the unattached state of the window, for example:

RVDEBUG(dhrystone) = @ARM920T_0:ARM-A-RR [Unattached]

When the window is attached to the current connection, this option is checked. The Code window title bar changes to reflect the new attachment, for example:

RVDEBUG(dhrystone) = @ARM920T_0:ARM-A-RR [Board]

**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 *, for example:

*ARM920T_0 ARM-A-R-RR ARM920T on localhost
In the active connections list, the connection that is attached to the Code window is marked with a checkmark . A connection might be marked, therefore, with both a checkmark and an asterisk to show that it is the current connection and that it is attached to the current Code window.

### Windows attachment

With a connection established, *attachment* refers to whether a window is tied to a particular processor. If a window is:

- **Attached**  
  It only displays information about the attached connection.

- **Unattached**  
  It only displays information about the current connection.

--- Note ---

If you are not licensed to work in multiprocessor debugging mode, all Code windows are unattached by default. However, this does not appear in the title bar.

Attachment enables you to use Code windows in a way that suits your working style, for example you might use a single unattached Code window and then cycle through the active connections displaying each current connection in turn. Alternatively, you might create multiple Code windows and attach each to a specified connection.

In multiprocessor mode, you can use a combination of attached and unattached Code windows as required. When a Code window is unattached, it displays information about the current connection and is often the first-choice window for the display of output messages from the debugger and the debuggee.

When a window is attached, the title bar shows what it is attached to:

- **[Board]**  
  Specifies that the window is attached to a connection, that is a debug target board or to a specified processor on a multiprocessor board.
  
  Toggle the **Attach Window to a Connection** option on the **Connection** menu to change this.

- **<blank>**  
  If the title bar does not contain **Unattached** then this window is attached to a chosen thread, that is it is always associated with this thread.
  
  Toggle the **Attach Window to a Thread** option on the **Thread** button drop-down to change this.

See *Attaching windows to multiple connections* on page 5-23 for details on configuring windows attachment.
Changing the current connection

The list of active connections is extended as you make connections to your targets. The last item on the list is the last connection made and RealView Debugger automatically identifies this as the current connection.

You can specify another connection to be the current connection, in the following ways:

- Click on the Connection button to cycle through the list of active connections until the required connection becomes the current connection.
- Click on the Connection button drop-down arrow to display the Connection menu, shown in Figure 5-9 on page 5-20. Select the required connection from the list of active connections. This connection then becomes the current connection.

Changing the current connection immediately updates all unattached windows. The new connection is shown in the title bar and the Color Box changes color to indicate the new connection.

Note

If you try to change the current connection using the Connection button from an attached Code window, you are warned that the window is attached, and given the option to detach the window or to cancel connection selection. If you do not wish to detach this Code window, you must create a new Code window and select the new connection in that.

Creating new windows

With multiple connections established, you can create new windows to display different views during your debugging session. Select View from the Code window menu to display the View menu, shown in Figure 5-10.

You can create new Code windows from the default Code window or from each new Code window as it opens. Each new Code window inherits its attachment from the calling window.
Attaching windows to multiple connections

This example describes how to connect to three different simulated targets and then to attach Code windows to each of these connections, ready to start debugging images.

Note

You can only work through this example if you are licensed to work in multiprocessor mode.

To attach windows to multiple connections:

1. Start RealView Debugger to display the default Code window (unattached).

2. Connect to three different target processors, as described in The Connection Control window on page 5-13. This example uses RVISS.

   As each target becomes the current connection, the title bar in the default Code window changes to reflect the new connection and the Color Box changes color. For example, making the last connection changes the title bar to:
   
   RVDEBUG = @SimARM_3:Sim [Unattached]

3. Select View → New Code Window from the default Code window main menu and open a second Code window, RVDEBUG_1. You can also open a new window using Alt+1.

4. Select View → New Code Window from the default Code window main menu (or press Alt+1) to open a third Code window, RVDEBUG_2.

   Arrange the windows on your desktop. The title bar of each Code window shows the current connection and the Color Boxes match.

With the Code windows displayed and the connections established, you can now attach each window to a specified connection:

1. Move the focus to the first Code window, RVDEBUG, and click on the Connection drop-down arrow to attach it to the current connection. Select Attach Window to a Connection.

   The title bar is updated to show that this window is now attached to the current connection, that is the last connection made:

   RVDEBUG = @SimARM_3:Sim [Board]

   The Color Box also changes color.

2. Move the focus to the second Code window, RVDEBUG_1. Click on the Connection button to cycle the current connection to the next one on the list.
Working with Multiple Target Connections

This action changes the current connection and updates the title bars of the two unattached windows, that is:

RVDEBUG_1 = @SimARM_1:Sim [Unattached]
RVDEBUG_2 = @SimARM_1:Sim [Unattached]

3. Attach the second Code window, RVDEBUG_1, to the current connection. Click on the Connection drop-down arrow and select Attach Window to a Connection. This action updates the title bar of the second Code window to show the attachment:

RVDEBUG_1 = @SimARM_1:Sim [Board]

4. Move the focus to the third Code window, RVDEBUG_2. Click on the Connection button to change the current connection. This action updates the title bar of the third Code window to show the new current connection:

RVDEBUG_2 = @SimARM_2:Sim [Unattached]

Because the other windows are now attached, only this Code window changes to display the new current connection.

5. Attach the third Code window, RVDEBUG_2, to the current connection. Click on the Connection drop-down arrow and select Attach Window to a Connection. This action updates the title bar of the third Code window to show the attachment:

RVDEBUG_2 = @SimARM_2:Sim [Board]

6. Move the focus to the first Code window, RVDEBUG.

7. Select View → Resource Viewer Window to display the Resource Viewer window. You can also press Alt+3.

8. Display the connection details, shown in Figure 5-11.

![Resource Viewer showing multiple connections](image.png)
The Resource Viewer window has inherited its attachment from the calling window and so the title bar and the Color Box reflect the default Code window. The Conn tab shows that SimARM_2 is the current connection by placing an asterisk at the left of the entry.

--- Note ---
You can display the Resource Viewer window from any Code window. Creating multiple instances of the Resource Viewer displays the same connection details as the calling window but each new instance is renamed, for example the second Resource Viewer window you display is named Resource Viewer_1.

With the connections established, you can move to each window in turn and load an image ready for debugging. See Working with projects on page 5-29 for details on attaching windows when working with projects.

Other ways to attach windows

The example in Attaching windows to multiple connections on page 5-23 describes one way to create windows and attach them to multiple connections. However, there are other ways to configure your views. Choosing the most efficient way to do this depends on your debugging environment. For example, another method would be to create the first Code window and then immediately attach it to the required connection. Each new Code window that you create is now attached to this connection by default. You can simply change attachment as you create each new window.

Changing windows attachment

Independent of the current connection, you can change the attachment of a window. With a Code window attached to a connection, click on the Connection button drop-down arrow and select a different connection from the active connections list. This immediately changes the Code window so that it is attached to the new connection, and updates the title bar (and Color Box) to show the new attachment. Changing windows attachment in this way does not change the current connection.

5.3.4 Disconnecting from targets

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

- the number and arrangement of active connections
- the number and attachment of Code windows
- which window has the focus when the disconnection option is used
- the state of processors and processes currently active and connected
- the required state of processors or processes following disconnection.
You can disconnect from the current target quickly, depending on the current debugging mode, for example:

- Select **File → Connection → Disconnect** from the Code window main menu. This immediately terminates the connection.

- Use the **Disconnect** button from the Connection group on the Actions toolbar. This immediately terminates the connection.

Code windows are not closed on disconnecting but their contents might change depending on the data they contain. For example any loaded images are unloaded and associated source files are closed, and entries displayed in a Register or Memory pane are cleared. This behavior depends on the update options you set for the window and the disconnect state of the target processor.

The options available are:

- **Disconnecting from the File menu**
- **Disconnecting from the Connection Control window** on page 5-27
- **Exiting RealView Debugger** on page 5-28
- **Setting disconnect mode** on page 5-28.

### Disconnecting from the File menu

When working with multiple targets, you can disconnect from the current connection, that is the connection visible in an unattached window, by selecting **File → Connection → Disconnect** from the Code window main menu. If you have attached the Code window to a connection, then this becomes the current connection. Disconnecting this way has the following effects:

- The current connection is terminated immediately.
- All active connections lists are updated.
- Any windows attached to the current connection are unattached.
- The next connection in the active connections list becomes the current connection.
- Title bars and Color Boxes for all unattached windows are updated to reflect the new current connection.
- Any windows already attached to other connections in the active connections list are not affected by terminating the current connection unless their parent connection becomes the new current connection. In this case, the title bars are updated to show that they are now attached to the current connection.
If the menu you select is in a window whose title bar does not show the current connection, that is, the window is attached to another connection in the active connections list, then this disconnects the attached connection. This has the following results:

- All active connections lists are updated.
- Any windows attached to the connection chosen for termination are unattached.
- Title bars and Color Boxes for all newly-unattached windows are updated to reflect the current connection.

If the connection chosen for termination is the only remaining connection, then this is the same as disconnecting in single processor mode, see the chapter describing connecting to targets in RealView Debugger v1.7 Target Configuration Guide for details. In multiprocessor mode, however, the Connection button is disabled on terminating the last connection.

To close any unwanted windows, select File → Close Window from the main menu. Remember that if you close all your Code windows, RealView Debugger exits.

**Disconnecting from the Connection Control window**

At any point in your debugging session, you can disconnect from a target using the Connection Control window. Use this method to specify which connection from the list of active connections is terminated. You can also disconnect several connections in turn or disconnect all connections in one step.

You can disconnect from a debug target in three ways:

- Double-click on a connection entry.
- Click the check box for a required entry so that it is unchecked.
- Right-click on a connection entry and select Disconnect from the Disconnection menu.

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

- The current connection is terminated immediately.
- All active connections lists are updated.
- Any windows attached to the current connection are unattached.
- The next connection in the active connections list becomes the current connection.
Title bars and Color Boxes for all unattached windows are updated to reflect the new current connection.

Any windows already attached to other connections in the active connections list are not affected by terminating the connection unless their parent connection becomes the new current connection. In this case, the title bars are updated to show that they are now attached to the current connection.

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

- The chosen connection is terminated.
- All active connections lists are updated.
- Any windows attached to the connection chosen for termination are unattached.
- Title bars and Color Boxes for all newly-unattached windows are updated to reflect the current connection.

If the connection chosen for termination is the only remaining connection, then this is the same as disconnecting in single processor mode. See the chapter describing connecting to targets in RealView Debugger v1.7 Target Configuration Guide for details. In multiprocessor mode, however, the Connection button is disabled on terminating the last connection.

**Exiting RealView Debugger**

The quickest way to close all connections is to exit the debugger. Choosing this method enables you to:

- terminate all connections
- exit the debugger but maintain connections for future sessions
- exit the debugger but restart with the same connections at your next debugging session.

For full details on these options, see the chapter describing exiting RealView Debugger in RealView Debugger v1.7 Essentials Guide.

**Setting disconnect mode**

You can control the way a processor is left when a connection is terminated. For example, you might want to exit RealView Debugger but leave the target running ready to reconnect at your next debugging session. This is useful when debugging multiprocessor systems or multithreaded applications.
See the chapter describing connecting to targets in *RealView Debugger v1.7 Target Configuration Guide* for full details on using the **Disconnection** menu from the Connection Control window and specifying the state of a processor after disconnection.

**Disconnecting all connections**

At any point in your debugging session, you can disconnect from all connected debug targets using the Connection Control window. Right-click on a top-level entry, for example the **ARM-ARM-NW** vehicle, to see the **Vehicle** context menu, shown in Figure 5-12.

![Figure 5-12 Vehicle context menu](image)

**Note**

The **Vehicle** context menu contains different options depending on the vehicle you choose from the Connection Control window.

In multiprocessor mode the **Vehicle** context menu contains the option **Disconnect All**. Selecting this option terminates all connections and has the following results:

- All connections are terminated.
- All windows are unattached.
- Title bars and Color Boxes for all windows are updated to reflect that there is no connection.
- The **Connection** button is disabled.

In multiprocessor mode the **Disconnect All** option is available when you have at least one connection to a debug target.

### 5.3.5 Working with projects

If you are licensed to work in multiprocessor debugging mode, you can work with:

- multiple connections (possibly containing RTOSs)
- multiple Code windows
- multiple projects
- different windows attachment.
This section describes:

- Working with multiple connections
- Working with attached windows on page 5-31
- Project binding with multiple connections on page 5-32
- Connecting and disconnecting on page 5-33.

Note

This section assumes that you are familiar with the concepts and terms explained in RealView Debugger v1.7 Project Management User Guide.

Working with multiple connections

Where you are working with multiple projects in multiprocessor debugging mode, the project environment depends on:

- your connections
- the order in which projects open
- project binding
- open windows and their attachment.

If you are licensed to work in multiprocessor debugging mode, project operations are relative to the current connection. This means that:

- When it first opens, the default Code window is unattached.
- The active project is shown in the default Code window title bar. If you are not connected, this is the last project that you open. When the project opens, the title bar shows the project name in angled brackets for example <dhrystone>. If you are connected, the active project is the last project that binds successfully. When the project binds, the title bar shows the project name in round brackets for example (dhrystone).
- The active project is shown as bound or unbound, in the default Code window title bar, depending on the current connection.
- Each active connection can have a different project bound to it. The bound project is the active project for that connection.
- The same project can bind to multiple connections as long as they correspond. This means a project might be the active project across multiple targets.
- The Project Control dialog box works across all open projects and all active connections, shown in Figure 5-13 on page 5-31.
Working with Multiple Target Connections

Select Project → Project Control... to use this dialog box to unbind or rebind projects in the usual way. See the chapter describing managing your projects in RealView Debugger v1.7 Project Management User Guide for details.

- If it is visible, the Process Control pane shows details for the project that is bound to the current connection.

Working with attached windows

When you are working with multiple connections, project operations are relative to the current connection and depend on windows attachment. This means that

- By default, the active project is shown at the top of the open project list (see Figure 5-13).

- If you are connected, an unattached Code window shows the current connection. It gives you direct access to the active project using the Project or Tools menus. The active project is the project bound to the current connection. If there is no project bound to the current connection, then the active project is the default.

- If you are connected, an attached Code window shows the attached connection. It gives you direct access to the active project using the Project or Tools menus. The active project is the project bound to the attached connection. If there is no project bound to the attached connection, then the active project is the default.

- You can use the Project Control dialog box to change the active project regardless of the attachment of the calling Code window.

- A new Code window inherits the project environment from the calling window. Therefore, a new Code window inherits the active project from the calling window.
Project binding with multiple connections

When you are working with a single project and multiple connections, project binding rules apply as explained in the chapter describing binding in RealView Debugger v1.7 Project Management User Guide.

To see an example of default binding:

1. Start up RealView Debugger.
2. Connect to two target processors, for example an ARM core that is part of your target hardware and a simulated ARM core using RVISS. The simulated ARM core becomes the current connection.
4. Select View → New Code Window to display a third Code window, RVDEBUG_2.
5. Use the Connection drop-down to attach the default Code window, RVDEBUG, to the connection to the ARM core. The title bar shows, for example:
   \[ RVDEBUG = @ARM940T_0:ARM-A-RR \]  
6. Attach the second Code window, RVDEBUG_1, to the connection to the simulated ARM core. The title bar shows, for example:
   \[ RVDEBUG_1 = @SimARM_2:Sim \]  
7. Leave the third Code window, RVDEBUG_2, unattached. The title bar shows the current connection, for example:
   \[ RVDEBUG_2 = @SimOAK_6:Sim \]  
8. Connect to another target processor for example an Oak DSP core. This becomes the current connection.
9. The third Code window, RVDEBUG_2, is unattached. The title bar shows the current connection, for example:
   \[ RVDEBUG_2 = @SimOAK_6:Sim \]  
10. Select Project → Open Project... from the default Code window main menu.
11. Load the required project, for example dhrystone.prj, into the debugger. This file is located in the \Examples directory in your root installation.
12. Display each Code window in turn and view the information in the title bar.

The open project matches only the ARM family of processors and so is bound by default to two connections:
There is only one open project and so this is the active project for all the connections. However, it is unbound in the title bar of the third (unattached) Code window, RVDEBUG_2.

If you move to the unattached Code window, RVDEBUG_2, and click on the Connection button to cycle to the next active connection, the title bar reflects the default Code window:

RVDEBUG_2(dhrystone) = @ARM940T_0:ARM-A-RR [Unattached]

You can cycle through the active connections in this way because this window, RVDEBUG_2, is unattached.

**Project binding with multiple projects**

If you now open a second project, for example Project_1, RealView Debugger tries to bind the project to a matching connection by default. In this case, the new project matches the ARM connections. Because the project currently bound to these connections is not autobound, RealView Debugger gives you the option to unbind the current project and bind the new project. This displays the list selection box showing the matching connections so that you can confirm the new binding.

--- **Note**  ---

If you click **Cancel**, the new project opens but the binding is unchanged.

If you open a third project, for example, the Oak project dtmf.prj, RealView Debugger repeats the binding procedure. In this case, there is no project bound to the Oak and so the new project binds by default. The Oak project is now the default active project and is shown in any unattached Code window title bar. The title bars of attached windows change to show the relevant active project.

**Connecting and disconnecting**

Connecting to and disconnecting from a debug target changes the project environment:

- If you open multiple projects and then connect to one or more targets, RealView Debugger binds any matching autobound projects first.
- If you open multiple projects where none specifies autobinding and then connect to one or more targets, RealView Debugger uses default binding to bind projects in the order specified by the open project list.
• If you open multiple projects and then connect to one or more targets, this changes the contents of all unattached windows.

• If you disconnect and there is a project bound to the connection, then the project unbinds but any close commands are not run because the connection has been lost.

• If you disconnect from one of your targets, this changes the contents of all windows, attached and unattached.

• If you disconnect from one of your targets, this might change the active project depending on the current project environment.

• If you disconnect, this does not close any open projects. Therefore, you can continue to make changes to the project properties. This applies to user-defined projects and auto-projects.

5.3.6 Using RealView ICE with multiple targets

The RealView ICE software enables you to debug multiple processors. It can also perform a synchronized start or stop of processors, for debugging multiprocessor systems where the processors interact with each other (see Processor execution synchronization on page 5-45 for details of working this way).

If you are debugging multiple processors, changes to configuration items in the Debug tab, shown in the Code window Register pane, do not replicate to the Debug tabs for the other processors on your target. The Debug tabs continue to show the old values until these processors stop. This forces RealView Debugger to update the window and refresh the display. However, if you change configuration items this takes immediate effect in the RealView ICE interface unit for all the processors in your system.

You can use RealView ICE to debug a wide range of ARM-based targets. The RealView ICE software translates debugger commands, for example to start or stop the processor, into JTAG control sequences for the chosen target.

As explained in Using target debug interfaces on page 5-11, some multiple target connections cannot be implemented using only one RealView ICE interface unit. When connected to multiple processors, the connection properties inherent in the interface apply to all the target processors, shown in the Connection Control window in Figure 5-14 on page 5-35.
Figure 5-14 RealView ICE connection in the Connection Control window

In Figure 5-14, the RealView ICE entry has been expanded to show the two processors on the target board. The connection properties defined for this connection apply to both processors. See RealView Debugger v1.7 Target Configuration Guide for more details on using the Connection Control window and on configuring your debug targets.

Select Programs → ARM → RealView ICE from the Windows Start menu to see the release notes that accompany your installation. These contain full details of all the processors supported by RealView ICE. See also RealView ICE User Guide for more details on how to configure multiple targets.

——— Note ———

You can use RealView ICE to connect to a target that incorporates DSP hardware with a suitable JTAG configuration. However, at this stage of development, it is not possible to connect to DSP cores using RealView ICE.

5.3.7 Using Multi-ICE with multiple targets

As explained in Using target debug interfaces on page 5-11, some multiple target connections cannot be implemented using only one Multi-ICE interface unit. When connected to multiple processors, the connection properties inherent in the interface apply to all the target processors, shown in the Connection Control window in Figure 5-15 on page 5-36.
Working with Multiple Target Connections

In Figure 5-15, the Multi-ICE entry has been expanded to show the two processors on the target board. The connection properties defined for this connection apply to both processors. See RealView Debugger v1.7 Target Configuration Guide for more details on using the Connection Control window and on configuring your debug targets.

You can use Multi-ICE to debug all ARM-based targets. The Multi-ICE software translates debugger commands, for example to start or stop the processor, into JTAG control sequences for the chosen target.

Configuring Multi-ICE

It is recommended that you turn off the cache mechanism in Multi-ICE when debugging multiple processors:

1. Select File → Connection → Connect to Target... to display the Connection Control window.
2. Right-click on the Multi-ICE entry and select Configure Device Info... from the context menu.
   The Multi-ICE DLL configuration dialog is displayed.
3. Click the Advanced tab.
4. Ensure that the Start-up with cache enabled check box is not selected, and click OK.

   **Note**
   Do not configure the board file when the debugger is connected to a target.
5.4 Display coherency

This section describes how RealView Debugger is affected by multiprocessor debugging. It is split into the following sections:

- Resource sharing and debugger consistency
- Saving and restoring your .brd file on page 5-38
- Defining shared memory regions on page 5-38.

If you are debugging a multithreaded application, this is not necessary provided that you are running in HSD or RSD mode.

5.4.1 Resource sharing and debugger consistency

Multiple processor systems normally include communication facilities so that each of the processors in the system can communicate with the others. Occasionally the communication is performed in the analog domain, but normally it is effected using one of the following digital methods:

- Shared memory, with either multiple ports or bus-sharing for each of the processors. When the whole memory and I/O space is shared and the same kind of processor is used, this is a Symmetric Multiprocessor (SMP).
- Point-to-point data links, using serial or parallel interfaces on each processor bus.
- Broadcast data links, such as 10Mbit Ethernet.

Systems that use point-to-point or broadcast data links do not normally share resources. However, when resources (such as memory) are shared between different processors, and RealView Debugger is attached to several processors, the debugger must ensure that data presented to the user is consistent.

For example, consider a session where you have two connections to two processors that share a region of memory. If you change a shared value using one connection, RealView Debugger has two options:

- Ensure that the change affects all connections accessing the shared memory.
- Provide a command that ensures that a given connection truly represents the current target state, for example, by reading everything again. This enables you to execute this command as required.

RealView Debugger provides a mechanism whereby changes to a given connection cause an update for each related connection. It does this by enabling you to define shared memory areas for different processors as part of your target configuration.
settings. Therefore, when a change for one connection affects the shared resource, the other connections are prompted to check whether they must update their window. You can also request updates as required using the pane context menus.

5.4.2 Saving and restoring your .brd file

In these examples you are amending your board file. This is normally stored in your RealView Debugger home directory. By default, the board file information is stored in rvdebug.brd. Other configuration files have extensions such as *.cnf and *.rbe.

You are recommended to backup this directory before starting the examples described in this chapter, so that you can restore your original configuration later. To do this:

1. Exit RealView Debugger.
2. Use Windows Explorer to display \home\user_name\, or the equivalent folder if you have not installed the product in the default location.
3. Right-button select the user_name folder icon and click Copy.
4. Click Paste, creating a new folder called Copy of user_name.

If you have to restore your board file:

1. Exit RealView Debugger.
2. Use Windows Explorer to display \home\user_name\, or the equivalent folder if you have not installed the product in the default location.
3. Right-button select the user_name folder and click Delete. Click OK to dismiss the check dialog.
4. Rename the folder Copy of user_name to user_name.

You can now restart RealView Debugger with your saved configuration.

If you want to return to the factory settings, delete the user_name folder and restart RealView Debugger. It creates a new default configuration for you.

5.4.3 Defining shared memory regions

This information is defined using the Connection Properties window where you configure the Advanced_Information block for each processor sharing memory. To do this:

1. Display the Connection Control window, and click on the Connect tab to see the available connections.
2. Right-click on the first processor sharing memory and select **Connection Properties...** from the context menu. This displays the Connection Properties window. The selected connection is highlighted in the left pane and the contents of this entry are in the right pane.

3. Select the **BOARD=** entry defining your target. Normally this is stored in a *.bcd file.

   **Note**
   Setting up and modifying **BOARD** entries is described in the configuring custom targets chapter in *RealView Debugger v1.7 Target Configuration Guide*.

4. Expand the following groups in turn:
   a. **Advanced Information**
   b. **Default**
   c. **Memory_block**.

5. Expand the **Default** group by double-clicking on the entry in the right pane. Your Connection Properties window should look like Figure 5-16.

   ![Figure 5-16 Defining shared memory regions](image)

   You use the **Memory_block** group to define areas of memory that have specific characteristics, one of which is whether the memory region is shared between this processor and another. Expand the **Attributes** group to see the settings **Shared** and **Shared_id**. The following examples show how you use them:

   - **Defining memory for a symmetric multiprocessor** on page 5-40
   - **Defining memory for a three processor multimedia system** on page 5-40.
Defining memory for a symmetric multiprocessor

A simple example is an SMP environment, in which two processors share all memory and all peripherals. To do this, change the settings for each processor (shown in the example in Figure 5-16 on page 5-39) like this:

1. Set the value of Start = 0.
2. Set the value of Length = 0xFFFFFFFF.
3. Expand the Attributes group.
4. Right-click and set the value of Shared = direct.
5. Set the value of Shared_id = 0x1.

**Note**
The actual value of a share ID is not relevant. You can use any small integer provided the same value is associated with each block sharing the same memory area.

Defining memory for a three processor multimedia system

A more complex example configuration, shown in Figure 5-17 on page 5-41, shows the address spaces of three processors sharing two memory regions.

Each entry in the Advanced_Information section of the board file describes the memory layout of a processor as one or more segments. For each processor and for each segment, the board file must include:

- the base memory address and length
- the type of memory, that is read-only, or read/write
- whether the segment is shared, and if so the share ID.

The details of the settings that you must make in your Connection Properties window to configure the three processors are shown in Figure 5-17 on page 5-41. You must start by either:

- editing the settings of the connection in the board file directly, for example the CONNECTION group for the Multi-ICE connection shown in Figure 5-16 on page 5-39
- creating a Board/Chip definition file, similar to the AP.bcd shown in Figure 5-18 on page 5-41, and then reference it from the board file using the BoardChip_name setting in the CONNECTION= group.
Figure 5-17 Example of a shared memory configuration

Using the BoardChip_name method makes the configuration more flexible, but it is slightly more complex to set up.

Figure 5-18 Editing the memory block in the AP Board/Chip definition file
To configure RealView Debugger for the target shown in Figure 5-17 on page 5-41, you must set up several memory blocks. See RealView Debugger v1.7 Target Configuration Guide for more information about setting up target configurations.

Each processor has a memory block for its private area and a block for a shared communication area. The ARM920T™ core has two shared areas, so it has three memory blocks.

You must add memory description blocks to the board Advanced Information group as follows:

**ARM966EJ-S_2**

The Memory block for this processor contains two sub-blocks:

- **LocalMem**
  - You must enter:
    - Start=0
    - Length=0x10000
    - Description="Local program memory"

- **GfxMem**
  - You must enter:
    - Start=0x10000
    - Length=0x10000
    - Description="Frame Buffer"
  - You must enter the following in the Attributes group:
    - Shared=direct
    - Shared_id=1

**ARM920T_0**

The Memory block for this processor contains three sub-blocks:

- **LocalMem**
  - You must enter:
    - Start=0
    - Length=0x40000
    - Description="Local program memory"

- **GfxMem**
  - You must enter:
    - Start=0x40000
    - Length=0x10000
    - Description="Frame Buffer"
  - You must enter the following in the Attributes group:
    - Shared=direct
    - Shared_id=1

- **CommsMem**
  - You must enter:
**Working with Multiple Target Connections**

Start=0x80000
Length=0x1000
Description="Shared IPC memory"
These Attributes are required to set up the sharing:
Shared=direct
Shared_id=2

**ARM940T_1**

The Memory_block for this processor contains two sub-blocks:

**LocalMem**
You must enter:
Start=0
Length=0x40000
Description="Local program memory"

**CommsMem**
You must enter:
Start=0x80000
Length=0x1000
Description="Shared IPC memory"
You must enter the following in the Attributes group:
Shared=direct
Shared_id=2

In this example, the memory map for each processor is defined using the device name (for example, ARM920T) followed by an underscore and the TAP controller ID for that processor (for example, _0). Including the TAP number as well as the processor name enables you to specify the exact processor even if you are using more than one of the same type of processor.

Within each processor memory block, some common properties of processor memory are defined, for example, defining the bus width using Access_size. These properties are inherited by the other memory specification blocks.

Specific properties, including start address and length of the memory regions, are defined for each of the memory regions. The regions are called LocalMem, CommsMem and GfxMem. In this example, the CommsMem region appears at the same place in the memory map of each of the processors accessing it, but the GfxMem does not. Where a shared region appears, in a given processor memory map, it is a function of the hardware memory address decoders on the target. It does not matter to RealView Debugger whether the shared regions map to the same addresses or to different addresses on the processors sharing them.
The default memory sharing state is unshared (indicated by the entry Shared=none), so the LocalMem definition does not have to state this. However, the CommsMem and GfxMem regions are shared, so the two attributes Shared and Shared_id must be specified for both regions. The value of Shared is one of:

- none The memory region is not shared.
- direct The memory region is shared.

--- Note ---

At this stage of development, the indirect option is not supported in RealView Debugger.

---

The memory region share IDs used in this example are:

1 the video buffer memory
2 the interprocessor communications memory.

There is nothing special about these particular values.

--- Note ---

Because the shared resources are described as part of the processor memory map, not by physical device, although there is normally only one shared memory device, RealView Debugger requires that the shared memory device is described at least twice, once for each processor sharing it.

---

**Using target debug interfaces**

As explained in Resource sharing and debugger consistency on page 5-37, the multiprocessor configurations described in this section, cannot be implemented using only one RealView ICE, or Multi-ICE, interface unit. When connected to multiple processors, the connection properties inherent in the interface apply to all the connections. See Using target debug interfaces on page 5-11 for more details.

The configurations described can be achieved using:

- multiple simulator connections using RVISS
- multiple debug target interfaces, for example RealView ICE units, or by mixing RealView ICE and Multi-ICE units.
5.5 Processor execution synchronization

When you have multiple processors that are cooperating within a single application, it is sometimes useful to be able to start all processors or to stop all processors with a single debugger command. RealView Debugger includes facilities for cross triggering and synchronizing processors:

- Start execution
- Stop execution
- Single-stepping.

This section describes how you can do this and what limitations exist:

- About execution synchronization
- Synchronization facilities on page 5-51.

5.5.1 About execution synchronization

When several processors are operating as part of a system, you might want to examine the state of all processors at one point. RealView Debugger does not synchronize processor activity unless it is told to do so. A processor only stops because you told the debugger to stop it, or because it triggered a breakpoint, or because the target operating environment stopped it. This section describes:

- Terms
- Synchronization and cross-triggering on page 5-50
- Synchronization, stepping and skid on page 5-51.

Terms

The following terms are used in this section:

Processor group

Within this section, the term processor group is used to refer to the set of processors that are configured to operate in a synchronized way.

Skid

For a processor group, skid is the time delay between the first processor stopping and the last processor stopping.

A processor group skids if one processor stops earlier than one or more of the others. This can result from differences in the way the processors are connected, different processor architectures, different instructions being executed, or because the debugger cannot issue the stop request concurrently.

Figure 5-19 on page 5-46 shows three processors stopping in response to an external event, such as clicking a stop button.
Skid means that, although stopping the processors is synchronized, they never stop at the same time. Table 5-1 shows typical values for the example shown in Figure 5-19.

In any multiprocessor system, the communication protocols between the processors must enable for differences in execution speed, and so this type of skid is not normally a problem.

### Table 5-1 Key of delay times for a user halt

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>1ms approx</td>
<td>Time for the debugger to process the user request.</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1ns...10ms approx</td>
<td>Time for the interface hardware to action the request, either in parallel or in sequence. The speed depends on whether this is performed using hardware or software.</td>
</tr>
<tr>
<td>$t_3$</td>
<td>1...10 clocks</td>
<td>Time for the processor to stop (normally, time for processor to reach next instruction boundary).</td>
</tr>
</tbody>
</table>
Loose synchronization

In both hardware and software environments, RealView Debugger can synchronize processors loosely. This is characterized by a large skid, of as much as several seconds (many million instructions), because of the way the debugger is connected to the processors. A large skid might also arise because there is no hardware synchronization control, so the debugger must issue stop commands manually.

Tight synchronization

In a hardware environment, RealView Debugger uses a closely synchronized system where this is supported by the underlying processor or emulator. This has a very short skid, usually a few microseconds or less, and perhaps only a single instruction.

Cross-triggering

Cross-triggering occurs when one processor in a group stops due to an internal or an external event, and this then causes other processors to stop. Cross-triggering differs from synchronization:

- Cross-triggering means that, if CPU1 hits an undefined instruction or triggers a breakpoint, that causes it to stop, CPU0 and CPU2 also stop as a result.
- Synchronization means that clicking the Stop button causes this action to be applied to all processors in the processor group.

See Synchronization and cross-triggering on page 5-50 for more details. The processor that initiated the stop, stops almost immediately, but others can take longer. If there is cross-triggering hardware on the target, a sequence similar to Figure 5-20 on page 5-48 occurs.
The initial stop activates an external signal on the processor, for example **DBGACK**, that causes the cross-triggering hardware to generate an input signal to the other processors, for example **CPU0 stop**, that stops the processors. Each of these other processors skids as it stops, as for a single processor system.

For a target system that does not have hardware cross-triggering, the debugger can perform a similar function in software. However, the processes involved are more complex, and the skid time is much longer. For example, hardware cross-triggering might be able to stop all processors five target instructions after the initial breakpoint. A software solution might take a million target instructions.

The sequence required for software cross-triggering is shown in Figure 5-21 on page 5-49.
Figure 5-21 Breakpoint stopping skid using software synchronization

The delays involved in this sequence are explained in Table 5-2. The figures for duration are for general guidance only.

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0...3 instructions</td>
<td>Time for breakpoint to stop processor</td>
</tr>
<tr>
<td>$t_2$</td>
<td>25...100ms approx</td>
<td>Time for debugger to notice processor stopped</td>
</tr>
<tr>
<td>$t_3$</td>
<td>50ns approx</td>
<td>Time for debugger to react to CPU1 stopping</td>
</tr>
<tr>
<td>$t_4$</td>
<td>50ns approx</td>
<td>Time to work out that a cross-triggering event occurred and which group of processors must be stopped</td>
</tr>
<tr>
<td>$t_5$</td>
<td>1...1000ms approx</td>
<td>Time for debugger to use the target debug interface to request the processors to stop, either in parallel or in sequence</td>
</tr>
<tr>
<td>$t_6$</td>
<td>1..10 instructions</td>
<td>Time for processor to stop (normally, time for processor to reach next instruction boundary)</td>
</tr>
</tbody>
</table>
Synchronization and cross-triggering

Synchronization applies equally to starting processor groups as well as stopping them, although starting a processor is easier to arrange and faster to do.

Having a target with closely synchronized processors and a short skid enables you to stop the system and be fairly sure that the overall state is as consistent as it was when you requested the stop. For a loosely synchronized system, whether the overall state is consistent once it has stopped is more dependent on the software and hardware architecture.

The actual length of skid varies and depends on many conditions. For example:

- If the stop request happens because one of the processors cross-triggers another, then the breakpointed processor has already stopped, but the debugger might not have registered that it must stop the other processors.

  This form of skid can be reduced by linking the processors together directly in hardware, so that one processor hitting a breakpoint stops other processors without debugger intervention.

- If one or more processors are controlled using debug monitor software, then the skid of that processor depends on whether the current task is interruptible or not.

- If one or more processors in the group share a memory bus, for example with a DMA controller, then another bus master can claim the bus and prevent the processor completing an instruction, so preventing it entering debug state.

- If the debugger must issue separate stop requests to each processor, then the host operating system might deschedule the debugger task between two of the stop requests and so introduce a significant extra delay.

It is normal that a multithreaded application is designed to be tolerant of differing execution speeds and differing execution orders for each of the constituent processes. In this case, communication attempts between processors are guarded to ensure data consistency. This is particularly true when the processors in a group run at differing clock speeds or using differing memory subsystems.

If communication guarding is not done, normal perturbations in the execution order might cause the application to fail. In communication systems that do not include very short communication timeouts, it is often possible to stop only one processor in a group. The other processors come to a halt through lack of, or excess of, data. Alternatively, you can let them continue to write to intentionally-overwriting communication buffers while you work.
--- Note ---

When working with a processor group, RealView Debugger warns you if the synchronization of the processors is loosely coupled by displaying the message:

Synchronization will be loose (intrusive) for some or all connections in the synch group.

---

**Synchronization, stepping and skid**

Synchronization applies equally to stepping processor groups as well as stopping and starting them. Having a target with closely synchronized processors enables you to step through the code to examine, for example, memory or registers on different processors.

Be aware, however, skid means that synchronized stepping might not behave in a predictable, or expected, way. Even where the code on the processors is identical, stepping moves to different locations on each core. If you are stepping at the disassembly level, RealView Debugger executes a single instruction and so, with short skid, the processors are more closely synchronized. However, stepping behavior is still unpredictable. Even where the processors are synchronized in hardware, there is a discrepancy of a few instructions.

**5.5.2 Synchronization facilities**

The synchronization facilities of RealView Debugger are accessed using the Synch tab on the Connection Control window. For full details see:

- *The Synch tab*
- *Execution controls* on page 5-52
- *Synchronizing processors* on page 5-53
- *Cross-triggering controls* on page 5-53
- *Working with synchronized processors* on page 5-55.

**The Synch tab**

Select **File → Connection → Synchronization Control...** to display the connection window with the *Synch* tab visible, shown in Figure 5-22 on page 5-52.
The top-level entries in the left pane list all available connections, with a check box beside each entry. This check box shows whether or not the processor, for that connection, is synchronized in any way. In the unchecked state, the processor is not affected by any other processor. In the checked state, the processor might be affected by other processors, depending on other controls. For example, to synchronize the ARM920T and the ARM940T processors, click on the check box associated with each entry. This immediately updates the Connection Control window with details of the type of synchronization, that is Loosely coupled or Tightly coupled.

Execution controls

Beneath the processor connection entries, the Execution controls define which operations are synchronized. On first opening the window, these are all checked by default. Execution controls can be set singly or in combination:

Step       The processor group is synchronized on step instructions, that is single stepping one processor also steps all other processors in the group.

------ Note ------
See Synchronization, stepping and skid on page 5-51 for more details on using this control.

Run        The processor group is synchronized on run instructions, that is starting one processor also runs all other processors in the group.

Stop       The processor group is synchronized on stop instructions, that is stopping one processor also stops all other processors in the group.

For example, if you want to stop and start your processors together, but are content to single-step each processor individually, you would check Stop and Run but not Step.
Synchronizing processors

Figure 5-23 shows an example Connection Control window with a group of two synchronized processors and their controls. Click the check box associated with each entry to synchronize the group on step, run, and stop instructions, that is all the processors start together and each processor stops all the others if it hits a breakpoint. Also, if you click the Stop button all the processors stop.

--- Note ---
See Synchronization, stepping and skid on page 5-51 for more details on synchronizing a processor group on step instructions, shown in Figure 5-24 on page 5-54.

Figure 5-23 shows another example Connection Control window with a group of two synchronized processors.

![Figure 5-23 Connection Control window Synch tab showing synchronized processors](image)

Here, the processor group is synchronized on run instructions only, that is each processor stops all the others if it hits a breakpoint. All the processors start together, but if you click the Stop button only one processor stops.

Cross-triggering controls

Expand the processor entries on the Synch tab to see the Trigger controls, shown in Figure 5-24 on page 5-54.
The Trigger controls describe communications between the specified processor and other processors in the group:

**In**
Select the check box to specify that the processor responds to the stop requests of other processors in the group.

**Out**
Select the check box to specify that, when the processor stops, it broadcasts a stop request to other processors in the group.

If a processor has both **In** and **Out** unchecked, that processor does not participate in cross-triggering. Indeed, if the processor check box is unchecked, the processor is not included in the group and so the state of the **In** and **Out** check boxes is irrelevant.

For example, if you want to prevent a processor from being stopped by another processor then you uncheck the **In** check box. You can still stop this processor if required, for example, by using a breakpoint.

The example shown in Figure 5-25 on page 5-55, consists of a group of two processors where cross-triggering has been set up to control the behavior of one processor based on the master processor.
Figure 5-25 Connection Control window Synch tab showing cross-triggering controls

In the configuration shown in Figure 5-25:

- the ARM920T core has Trigger Out enabled
- the ARM940T core has Trigger In and Trigger Out enabled.

When the ARM920T stops it broadcasts a stop request to the other processors in the group. The ARM940T responds to this signal and stops. However, if the ARM940T stops, its broadcast is ignored by the ARM920T and so the ARM920T processor does not stop.

With this system of controlling synchronization you can create both master-slave and peer-to-peer synchronization groups. However, you cannot create multiple independent processor groups, that is where two sets of processors are synchronized within the group but not between the two groups.

**Working with synchronized processors**

With the processor group controls set in the Synch tab, you can use a single, unattached Code window to view the connections, or set up multiple Code windows, and begin the debugging session.

If you are using multiple Code windows, it is recommended that you make one of the synchronized processors the current connection and that you attach a Code window to this connection as the first-choice window for displaying debugger messages.

Remember the following when working with synchronized processors:

- There is no difference in behavior when hardware cross-triggering or synchronization is available, although there is a large reduction in skid.
- There is no difference in behavior between simulators and other hard targets (boards) although a suitable bridging product is required to synchronize simulators.
Working with Multiple Target Connections
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 2-7 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. See the ARM MultiTrace User Guide for more information.

This appendix contains the following sections:
- ARM MultiTrace and ARM Multi-ICE on page A-2
- ARM RealView Trace and RealView ICE on page A-4
- Agilent 16600 or 16700 logic analyzer and Emulation Probe on page A-5
- Agilent 16600 or 16700 logic analyzer and Multi-ICE on page A-9
- Agilent Emulation Probe and Trace Port Analyzer (E5904B) on page A-12
- Tektronix TLA 600 or TLA 700 logic analyzer and Multi-ICE on page A-15.
A.1 ARM MultiTrace and ARM Multi-ICE

The ARM® MultiTrace™ analyzer is available from ARM Limited. When used with Multi-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.

A.1.1 Setting up

To set up your hardware to enable tracing in the RealView Debugger as shown in Figure A-1 on page A-3:

1. Connect and configure the Multi-ICE interface unit as described in the Using Multi-ICE with Debuggers chapter of the Multi-ICE User Guide.

2. Connect and configure the MultiTrace interface unit as described in the Getting Started chapter of the MultiTrace User Guide.

For information on connecting to and configuring targets for use with RealView Debugger, see the RealView Debugger v1.7 Target Configuration Guide.
Figure A-1 Connections for Multi-ICE and Multi-Trace using a separate Multi-ICE server
A.2 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-2, 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 RealView Debugger v1.7 Target Configuration Guide.

![Connections for RealView ICE and RealView Trace](image-url)

Figure A-2 Connections for RealView ICE and RealView Trace
A.3 Agilent 16600 or 16700 logic analyzer and Emulation Probe

To set up your hardware and enable tracing in the RealView Debugger, as shown in Figure A-3 on page A-6:

1. Set up either the Agilent 16600 or 16700 logic analyzer with the Agilent Emulation Probe, and an Agilent logic analyzer card supporting:
   - sampling rates at least as high as the core clock frequency of the target (at least twice as high if using the four-bit data port for the ETM)
   - a minimum of 21 signal inputs
   - a minimum of 10,000 words (samples) of memory.

2. Connect up all hardware as shown in Figure A-3 on page A-6. Power up all hardware except the ARM target board.
3. Configure the network setup using the user interface of the logic analyzer. Typically, the network settings are part of the system administration functionality that you can access by clicking **System Admin** in the Logic Analysis System window. See the logic analyzer documentation for more details.

4. Check that version numbers are correct for the following:

**Analysis system software**

Must be A.01.40.00 or later.
Processor support software
Must be A.01.40.00 or later.

Emulation Module firmware
Must be the following version numbers or later for the various components of the Emulation Module:
E3499B Series Emulation System
 Version: A.07.64
 Location: Generics
E3459B ARM7/9 JTAG Emulator
 Version: B.02.04
E3459Q ARM7/9 Trace Port Analyzer
 Version: Q.01.00.

To view the software versions, select the Software Install tab in the System Administration Tools window, and click List. If you require an upgrade for the software, contact Agilent technical support by following the instructions at the website http://www.agilent.com.

To view the firmware version, right-click on the Emulator icon (in the Logic Analysis System or Workspace window) and select the Update Firmware option to display the current version. If you require an upgrade:

b. Copy the files to the hard disk of the logic analyzer, placing them in the directory /hplogic/firmware/run_cntrl.
c. Click Update Firmware in the Update Firmware window. The upgrade occurs automatically.

Note: The Agilent website provides more details on this process.

5. Configure the analyzer software. During this process, you must record the following information so that you can set up RealView debugger to match:

• the number of target signals you are capturing, either wide (16-bit) or narrow (8-bit)
• the clock definition, either single edge or double edge.

The provided analyzer configuration files assume full rate (single edge) clocking, and no multiplexing or demultiplexing of the data. If you want to use half-rate clocking, multiplexing, or demultiplexing, you will have to modify the configurations that are loaded into the analyzer.
You can configure the analyzer in the following ways:

- **Click File Manager** in the Logic Analysis System window. Load in an appropriate generic configuration file. You can then save this back to a configuration file specific to the logic analyzer and appropriate slot.

  — **Note**
  
  The following logic analyzer configuration files are available:
  
  — **CARMETM_9**, corresponding to an 8-bit port width (with timestamps)
  — **CARMETM_10**, corresponding to an 8-bit port width
  — **CARMETM_11**, corresponding to a 16-bit port width (with timestamps)
  — **CARMETM_12**, corresponding to a 16-bit port width.

  Configurations using an 8-bit port width are also valid for use with a 4-bit ETM trace port.

  Contact Agilent to obtain a CD-ROM software update for logic analyzers. This update always contains the latest configuration files required for ETM tracing.

- **Click Setup Assistant** (if available) in the Logic Analysis System window. With this method, the process of loading a configuration file is split into a series of simple steps. For each step, you are prompted for information that allows the Setup Assistant to autogenerate a configuration file with your specifications. See the logic analyzer documentation for more details.

6. **Power up the ARM target board.** Your logic analyzer hardware is now configured for use with RealView Debugger.

7. **Configure your target in RealView Debugger to enable tracing** (see the *RealView Debugger v1.7 Target Configuration Guide*).
A.4 Agilent 16600 or 16700 logic analyzer and Multi-ICE

The difference between this configuration and the Agilent-only configuration is that the Agilent Emulation Module is replaced by the Multi-ICE interface unit.

To set up your hardware and enable tracing in RealView Debugger, you must:

1. Set up either the Agilent 16600 or 16700 logic analyzer with an Agilent logic analyzer card supporting:
   - sampling rates at least as high as the core clock frequency of the target (at least twice as high if using the four-bit data port for the ETM)
   - a minimum of 21 signal inputs
   - a minimum of 10,000 words (samples) of memory.

2. Connect a Multi-ICE interface unit as described in the Multi-ICE User Guide.

3. Connect the Multi-ICE JTAG cable between the interface unit and the JTAG plug on the target board, and connect the logic analyzer Mictor connector from ports Pod 1 and Pod 2 to the Trace port Mictor socket.
   If you do not have separate JTAG and Trace connectors on the target board, you must use an adaptor board plugged into the Trace connector. The board can be obtained from your logic analyzer supplier.

4. Connect up the rest of the hardware as shown in Figure A-4 on page A-11.

5. Power up all hardware except the target board.

6. Start the Multi-ICE server software on the host workstation and verify that autoconfiguration of the hardware works.

7. Configure the network interface of the logic analyzer. Typically, the network settings are part of the system administration functionality that you can access by clicking System Admin in the Logic Analysis System window. See the logic analyzer documentation for more details.

8. Check that version numbers are correct for the following:

   **Analysis system software**
   Must be A.01.40.00 or later.

   **Processor support software**
   Must be A.01.40.00 or later.

To view the software versions, select the Software Install tab in the System Administration Tools window, and click List. If you require an upgrade for the software, contact Agilent technical support by following the instructions at the website http://www.agilent.com.
9. Configure the analyzer software. During this process, you must record the following information so that you can set up RealView Debugger to match:
   - the number of target signals you are capturing, either wide (16-bit) or narrow (8-bit)
   - the clock definition, either single edge or double edge.
   The provided analyzer configuration files assume full rate (single edge) clocking, and no multiplexing or demultiplexing of the data. If you want to use half-rate clocking, multiplexing, or demultiplexing, you will have to modify the configurations that are loaded.
   
   You can configure the analyzer in either of the following ways:
   - Click **File Manager** in the Logic Analysis System window. Load in an appropriate generic configuration file. You can then save this back to a configuration file specific to the logic analyzer and appropriate slot.
     
     **Note**
     The following logic analyzer configuration files are available:
     - **CARMETM_9**, corresponding to an 8-bit port width (with timestamps)
     - **CARMETM_10**, corresponding to an 8-bit port width
     - **CARMETM_11**, corresponding to a 16-bit port width (with timestamps)
     - **CARMETM_12**, corresponding to a 16-bit port width.
     Configurations using an 8-bit port width are also valid for use with a 4-bit ETM trace port.
     Contact Agilent to obtain a CD-ROM software update for logic analyzers. This update contains the latest configuration files required for ETM tracing.
     
     **Note**
     If you use the Setup Assistant, you must select the **Setup Assistant** option in the **Logic Analyzer Configuration** dialog box when configuring RealView Debugger.
   - Click **Setup Assistant** (if available) in the **Logic Analysis System** window. With this method, the process of loading a configuration file is split into a series of simple steps. For each step, you are prompted for information that allows the Setup Assistant to autogenerate a configuration file with your specifications. See the logic analyzer documentation for more details.

10. Power up the ARM target board. Your logic analyzer hardware is now configured for use with RealView Debugger.

11. You must now configure your target in RealView Debugger to enable tracing (see the **RealView Debugger v1.7 Target Configuration Guide**).
Figure A-4  Agilent Trace Port Analyzer and Multi-ICE Version 2.2
A.5  **Agilent Emulation Probe and Trace Port Analyzer (E5904B)**

The Agilent integrated trace solution is supplied in the following parts:

- The Agilent Integrated EP and TPA (EP/TPA). This contains an *Emulation Probe* (EP), which contains network interface and control logic, and a *Trace Port Analyzer* (TPA), which provides signal capture logic and memory.
- The target buffer board, which buffers the signals for transmission to the TPA.

Figure A-5 shows the configuration using E5904B components.

---

**Note**

- Some target boards include a Mictor trace port connector and an IDC JTAG connector. The Mictor connector includes signals for the JTAG port, and these are routed on the Target buffer board to the 20-way ribbon cable. Do not connect a JTAG interface to both the Target buffer board and to the target IDC connector.
- ARM does not support the use of RealView Debugger and the Agilent EP/TPA with any other JTAG interface unit.
You cannot connect a JTAG controller to the JTAG control interface of the target when the JTAG interface of the Agilent EP/TPA is also connected using the Micror trace connector.

To set up the E5904B hardware to enable tracing in RealView Debugger:

1. Connect the Agilent EP/TPA to the local area network and to the target interface board. Connect the target interface board to the buffer board. These connections are shown in Figure A-5 on page A-12.

2. Connect up the power cables and switch on the power to the test equipment.

3. Configure the Emulation Probe with the correct network address, and check that the probe is running the correct version of the firmware. To do this:
   b. Set up a dumb terminal session on your workstation, using HyperTerminal for example. The serial port settings should be:
      - 9600 baud
      - 8 data bits
      - no parity
      - 1 stop bit
      - hardware handshake off.
      Power-cycle the probe. After about a minute, the prompt >p is displayed in the terminal window.
   c. At the >p prompt, type ver -a to check that the probe is running the correct firmware.

   Note
   It is strongly recommended that you upgrade your firmware to the latest versions of the firmware. To do this, follow the instructions provided on the website http://www.agilent.com.

   d. At the >p prompt, type lan to view the network configuration of your probe. The network configuration of your probe is displayed. Refer to the Emulation Probe documentation for details on this output.
   e. Set the network address assigned to the probe by typing:
      lan -i network-address
where `network-address` must be replaced with the dotted-quad network address assigned to the probe. You might also have to set the netmask (using `lan -s`) and default gateway (using `lan -g`), depending on the nature of the network. For more details on the `lan` command, type `help lan` or refer to the Agilent documentation.

The administrator for your network must assign a static name and network address to the device. You cannot use DHCP network addresses with the current firmware.

--- **Note** ---

After you have entered this command, you must power-cycle the probe for the change to take effect.

You might also have to change other network parameter settings using the `lan` command.

---

f. **Power-cycle the probe.** After a short while, the version information is displayed, as shown in step 3c.

You can now remove the RS-232 serial cable.

4. **Power up the target board.** Your EP/TPA hardware is now configured for use with RealView Debugger.

5. **Configure your target in RealView Debugger to enable tracing** (see the *RealView Debugger v1.7 Target Configuration Guide*).
A.6 Tektronix TLA 600 or TLA 700 logic analyzer and Multi-ICE

To set up your hardware to enable tracing in RealView Debugger, as shown in Figure A-6 on page A-16:

1. Install the Multi-ICE software on the analyzer, using the CD-ROM unit on the rear panel of the analyzer, as described in the Multi-ICE User Guide.

2. Set up the analyzer as described in the Tektronix TLA Logic Analyzer ARM ETM Support Package Instructions. In particular, connect the Mictor socket of a Tektronix P6434 Mass Termination Probe to the target board, and the two module ends of the cable to the analyzer.
   See the logic analyzer documentation for full details on these connections.

3. Connect the analyzer port to the Mictor trace connector on the target board.

4. Verify that the Dragonfly Software TLA COM Server application is running on the Tektronix analyzer.

5. Connect the Multi-ICE interface unit to the parallel port on the rear panel of the analyzer, and to the JTAG connector on the target board.

6. Power up the ARM target board.

7. Start the Multi-ICE server software using Start → Programs → ARM Multi-ICE v2.2 → Multi-ICE Server.

8. Select Auto-configure from the Multi-ICE server File menu and verify that the target board can be autoconfigured.

9. Your logic analyzer hardware is now configured for use with RealView Debugger. You must now configure your target in RealView Debugger to enable tracing (see the RealView Debugger v1.7 Target Configuration Guide).
Figure A-6 Tektronix TLA 700 analyzer and Multi-ICE
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 the following sections:

- **ARM MultiTrace and ARM Multi-ICE** on page B-2
- **Embedded Trace Buffer and ARM Multi-ICE** on page B-6
- **ARM RealView Trace and RealView ICE** on page B-10
- **ARM Multi-ICE for XScale** on page B-15
- **Agilent 16600 or 16700 Logic Analyzer and Emulation Probe** on page B-17
- **Agilent 16600 or 16700 Logic Analyzer and ARM Multi-ICE** on page B-24
- **Agilent Trace Port Analyzer and Agilent Emulation Probe** on page B-27
- **Tektronix TLA 600 or TLA700 and ARM Multi-ICE** on page B-30
- **Simulators using the Simulator Broker connection** on page B-33.

See the RealView Debugger v1.7 Target Configuration Guide for general information on connecting to targets.
B.1 ARM MultiTrace and ARM Multi-ICE

This section describes how to set up RealView Debugger to connect to a combination of ARM® MultiTrace™ and ARM Multi-ICE®. Refer to the Multi-ICE User Guide for more information about Multi-ICE, and to the MultiTrace User Guide for more information on setting up the MultiTrace unit.

To install and configure Multi-ICE and MultiTrace:

1. Install Multi-ICE on both the workstation you are debugging on and the workstation that the Multi-ICE interface unit connects to.
2. Install MultiTrace on the workstation you are debugging on.
3. Configure the Multi-ICE server on the workstation that the Multi-ICE interface unit is connected to.
4. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7 to start RealView Debugger.
5. In the Code window, select File → Connection → Connect to Target... to display the Connection Control window.
7. If Multi-ICE is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-1.

![Figure B-1 RDI Targets List](image)
c. If Multi-ICE is present, select the check box. If not, click Add DLL... and select Multi-Ice.dll from the Multi-ICE install directory. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close this dialog.

8. Right-click on the Multi-ICE entry and select Configure Device Info... from the context menu to display the ARM Multi-ICE Configuration dialog box (Figure B-2).

![Figure B-2 Multi-ICE configuration dialog box](image)

9. Depending on the location of the Multi-ICE server that you are using, click on either This computer or Another computer.

   If you select Another computer, in the subsequent dialog box type in the name of the remote workstation, or locate it using the tree control. Click on OK.

10. Select the ARM processor that you are tracing from the list shown in Available devices.

11. Optionally, enter your name in the Connection name text field. This is displayed in the Multi-ICE server window and can help if you are sharing the server with other users.

12. Click the Trace tab to make it visible (Figure B-3 on page B-4).
Setting up the Trace Software

Figure B-3 Multi-ICE configuration dialog box showing the Trace tab

The tab contains the controls required to configure the trace control software. The **Select a Trace Capture DLL...** drop-down list contains the names of the currently available trace capture drivers. These drivers read the trace information from the *Embedded Trace Macrocell™* (ETM) and translate it into the format required by the debugger.

13. Select the MultiTrace driver from the **Select a Trace Capture DLL...** drop-down list. This specifies the driver file that is used to control the trace capture device:
   
   - If `multitrace.dll` is present in the drop-down list, select it.
   - If `multitrace.dll` is not present in the drop-down list, click **Add...** and select `multitrace.dll` from the MultiTrace installation directory. (You might have to configure Windows Explorer to ensure that files with the extension `.dll` are not hidden from view.)

14. With `multitrace.dll` selected, the MultiTrace configuration controls are added to the Multi-ICE Trace tab (Figure B-4 on page B-5). Configure MultiTrace as described in the MultiTrace User Guide.
15. Click **OK** to exit the Multi-ICE dialog box.

16. Select Multi-ICE from the targets list in the RealView Debugger Connection Control window, by expanding the second-level **Multi-ICE** entry, and selecting the processor connection, such as **ARM966E-S**.

17. Close the Connection Control window.
B.2 Embedded Trace Buffer and ARM Multi-ICE

This section describes how to set up RealView Debugger to connect a combination of Embedded Trace Buffer and ARM Multi-ICE. Refer to the Multi-ICE User Guide for more information about Multi-ICE.

To install and configure Embedded Trace Buffer and Multi-ICE:

1. Install Multi-ICE on both the workstation you are debugging on and the workstation that the Multi-ICE interface unit connects to.
2. Configure the Multi-ICE server on the workstation that the Multi-ICE interface unit is connected to.
3. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7 to start RealView Debugger.
4. In the Code window, select File → Connection → Connect to Target... to display the Connection Control window.
5. Expand the top-level ARM-A-RR vehicle.
6. If Multi-ICE is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-5.

![Figure B-5 RDI Targets List](image)
c.  If Multi-ICE is present, select the check box. If not, click **Add DLL...** and select `Multi-Ice.dll` from the Multi-ICE install directory. (You might have to configure Windows Explorer to ensure that files with the extension `.dll` are not hidden from view.) Click **Close** to close this dialog.

7. Right-click on the Multi-ICE entry and select **Configure Device Info...** from the context menu to display the ARM Multi-ICE Configuration dialog box (Figure B-6).

![Figure B-6 Multi-ICE configuration dialog box](image)

8. Depending on the location of the Multi-ICE server that you are using, click on either **This computer** or **Another computer**.

   If you select **Another computer**, in the subsequent dialog box type in the name of the remote workstation, or locate it using the tree control. Click on **OK**.

9. Select the ARM processor that you are tracing from the list shown in **Available devices**.

10. Optionally, enter your name in the **Connection name** text field. This is displayed in the Multi-ICE server window and can help if you are sharing the server with other users.

11. Click the **Trace** tab to make it visible (Figure B-7 on page B-8).
Setting up the Trace Software

The tab contains the controls required to configure the trace control software. The Select a Trace Capture DLL... drop-down list contains the names of the currently available trace capture drivers.

12. Select the Embedded Trace Buffer driver from the Select a Trace Capture DLL... drop-down list. This specifies the driver file that is used to control the trace capture device:
   
   • If onchiptrace.dll is present in the drop-down list, select it.
   • If onchiptrace.dll is not present in the drop-down list, click Add... and select onchiptrace.dll from the Multi-ICE installation directory. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.)

13. With onchiptrace.dll selected, the configuration for the current processor is displayed in the Trace tab.
14. Click **OK** to exit the Multi-ICE dialog box.

15. Select Multi-ICE from the targets list in the RealView Debugger Connection Control window, by expanding the second-level Multi-ICE entry, and selecting the processor connection, such as ARM966E-S.

16. Close the Connection Control window.
B.3 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. Refer to the RealView ICE User Guide for more detailed information.

To install and configure RealView ICE and RealView Trace:

1. Install RealView ICE and RealView Trace and connect up the hardware as described in the chapter Using RealView Trace in the RealView ICE User Guide.

2. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7 to start RealView Debugger.

3. If the RealView ICE target vehicle and access provider nodes are not present, add the target vehicle and access provider nodes:
   b. Select the node at the root of the tree, that represents the board file.
   c. Right-click, and choose Make New Group from the context menu that appears, as shown in Figure B-9.
   d. In the Group Type/Name selector window, choose CONNECTION and set the Group name to the name that you want to use for the access provider node, such as RealView_ICE. Click OK. A new CONNECTION node appears in the Connection Properties window.
   e. Expand the new CONNECTION node, and click on Connect_with.
f. In the right-hand pane, select the Manufacturer node.

g. Right-click, and choose ARM-ARM-NW - RealViewICE from the context menu that appears, as shown in Figure B-10.

![Figure B-10 Setting how to connect](image)

h. Choose File → Save Changes to save the new configuration.

4. Expand the RealView ICE access provider node by double-clicking on it. If you see the error shown in Figure B-11, the target node is missing.

![Figure B-11 Error when the target node is missing](image)

If you see this error, click Cancel, and add the target node:

a. Open the Connection Control window by selecting File → Connection → Connect to Target from the Code window.

b. Select the relevant RealView ICE access provider node in the Connection Control window.

c. Right-click, and choose Configure Device Info from the context menu that appears. RealView ICE displays the error that is shown in Figure B-12 on page B-12.
d. Click Empty to create an empty configuration file. A dialog appears asking you where to store this file, as shown in Figure B-13.

![Figure B-13 Selecting where to store the new configuration file](image)

Figure B-13 Selecting where to store the new configuration file

e. Save this empty configuration file in your RealView Debugger home directory. Choose a filename to identify your RealView ICE interface unit. Click Save. The RVConfig dialog appears.

5. If the RVConfig dialog is not already open, open it in the following way:
   a. Selecting File → Connection → Connect to Target in the Code window to open the Connection Control window.
   b. Select the relevant RealView ICE access provider node in the Connection Control window.
   c. Right-click, and choose Configure Device Info from the context menu. The RVConfig dialog appears, as shown in Figure B-14 on page B-13.
6. Select the RealView ICE node in the left-hand pane and click Connect. A Devices node is added to the tree diagram.

7. 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.

8. Set the JTAG clock speed by selecting the clocking that you want in the area at the bottom of the RVConfig dialog. 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.
9. Set the **Vendor** property of the RealView ICE connection to ARM:
   a. Open the **Connection Properties** window by selecting **File → Connection → 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-15.
   
   ![Figure B-15 Connection Properties window show Logic_Analyzer settings](image)
   
   **Figure B-15 Connection Properties window show Logic_Analyzer settings**
   
   f. In the right-hand pane, right-click on **Vendor**, and select **ARM** from the context menu.
B.4 ARM Multi-ICE for XScale

This section describes how to set up RealView Debugger to connect to ARM Multi-ICE for XScale™. Refer to the Multi-ICE Installation Guide or the Multi-ICE User Guide for more information about Multi-ICE.

To install and configure Multi-ICE for XScale:

1. Install Multi-ICE on both the workstation you are debugging on and the workstation that the Multi-ICE interface unit connects to.

2. Configure the Multi-ICE server on the workstation that the Multi-ICE interface unit is connected to.

3. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7 to start RealView Debugger.

4. Select File → Connection → Connect to Target... to display the Connection Control window.

5. Expand the top-level ARM-A-RR vehicle.

6. If Multi-ICE is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-1 on page B-2.
   c. If Multi-ICE is present, select the checkbox. If not, click Add DLL... and select Multi-Ice.dll from the Multi-ICE install directory. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close this dialog.

7. Right-click on the Multi-ICE entry and select Configure Device Info... from the context menu to show the ARM Multi-ICE Configuration dialog box (shown in Figure B-2 on page B-3).

8. Depending on the location of the Multi-ICE server that you are using, click on either This computer or Another computer.
   If you select Another computer, in the subsequent dialog box type in the name of the remote workstation, or locate it using the tree control. Click on OK.

9. Select the ARM processor that you are tracing from the list shown in the list of Available devices.

10. Optionally, enter your name in the Connection name text field. This is displayed in the Multi-ICE server window and can help if you are sharing the server with others.
11. Click **OK** to exit the Multi-ICE dialog box.

12. Select Multi-ICE from the targets list in the Connection Control window, by expanding the second-level Multi-ICE entry, and selecting the processor connection, such as ARM966E-S.

13. Close the Connection Control window.
B.5 Agilent 16600 or 16700 Logic Analyzer and Emulation Probe

This section describes how to set up RealView Debugger to connect an Agilent Logic Analyzer and an Emulation Probe. You must have already set up the logic analyzer software as part of the hardware setup (see Appendix A Setting up the Trace Hardware).

Note

ARM Agilent Debug Interface (ADI) is not supplied with RealView Developer Suite, and must be purchased separately.

To configure the Agilent 16600 or 16700 analyzer:

1. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7.
2. Select File → Connection → Connect to Target... to display the Connection Control window.
4. If ADI is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-1 on page B-2.
   c. If ADI is present, select the checkbox. If not, click Add DLL... and select Gateway2.dll from \bin. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close the RDI Target List dialog.
5. Right-click on the ADI entry and select Configure Device Info... from the context menu to show the Gateway Configuration dialog box (Figure B-16 on page B-18).
6. Ensure that the **Connection Details** tab is displayed.

7. Fill in the **Network Details** for the Emulation Probe, and click **Lookup**.

8. Click on the **Advanced** tab (Figure B-17 on page B-19).
Figure B-17 Gateway Configuration dialog box, Advanced tab

9. If your target is running in big endian mode, click on Big-endian.

10. Click on the Trace tab (Figure B-18 on page B-20).
11. Specify the Gateway2 driver in the **Select a Trace Capture DLL...** drop-down list. This specifies the driver file that is used to control the trace capture device:
   - If `Gateway2.dll` is present in the drop-down list, select it.
   - If `Gateway2.dll` is not present in the drop-down list, click **Add...** and select `Gateway2.dll` from `\bin`. (You might have to configure Windows Explorer to ensure that files with the extension `.dll` are not hidden from view.)

The configured Gateway **Trace** tab is shown in Figure B-19 on page B-21.
12. In the System section of the dialog box, click on Logic Analyzer.

13. Enter the Network Address of the Agilent Logic Analyzer.

14. Click Configuration to display the Logic Analyzer Configuration dialog box (Figure B-20).
15. Select the appropriate startup option to indicate the level of initialization carried out by the debugger:

- Select the **Automatic** option if you want the debugger to ensure, at start-up, that the logic analyzer is fully initialized to carry out tracing. In this case, you must specify the appropriate Machine Name, Lister Name, and Config File in the dialog box.
  
  You must specify the full directory path to the configuration file.

- Select the **Set-up Assistant** option if you do not want the configuration file to be loaded by the debugger. The other parameters (Machine Name and Lister Name) are initialized with the given values. This mode is appropriate only when you are loading a configuration file from the logic analyzer user interface, which can be done using either of the following:
  
  — the Setup Assistant
  — the File Manager tool.

If you are using the default logic analyzer configuration files provided by Agilent, you must:

- set the machine name as **ARM ETM Analyzer**
- set the lister name as **ETM Data**.

**Note**

The default logic analyzer configuration files cannot be loaded directly by the analyzer. Instead, using the file manager tool on the analyzer user interface, you must load each file, save the configuration back to an appropriate filename, and specify this new name as the configuration to load.

16. Click **OK** successively to exit each of the dialog boxes.

17. In the RealView Debugger Code window, select ADI from the targets list in the Connection Control window, by expanding the second-level **Multi-ICE** entry, and selecting the processor connection, such as **ARM966E-S**.

18. Select **Tools → Analyzer/Trace Control → Configure Analyzer Properties** to display the Configure ETM dialog. Use the information you recorded when setting up the analyzer to set up the ETM as follows:

  a. Set the **Trace data width** to 16 bit for a wide analyzer connection, or to 8 bit or 4 bit (depending on the target hardware).
  
  b. Enable **Half-rate clocking** if you want to use double edge clocking.
  
  c. Click **OK** to accept the changes.

RealView Debugger is now configured for tracing.
For detailed information on setting up the Agilent 16600 or 16700 Logic Analyzer and Emulation Probe, see the Agilent documentation, which can be accessed from the website http://www.agilent.com. See Other publications on page x for details.

For detailed information on setting up and using ARM ADI, see the *ARM ADI User Guide*. 
B.6 Agilent 16600 or 16700 Logic Analyzer and ARM Multi-ICE

This section describes how to set up RealView Debugger to connect to a combination of Agilent Logic Analyzer and ARM Multi-ICE. You must have already set up the logic Analyzer software as part of the hardware setup (see Appendix B Setting up the Trace Software).

Note

ARM ADI is not supplied with RealView Developer Suite, and must be purchased separately.

Refer to the ARM ADI User Guide for details on setting up ARM ADI.

To install and configure the Agilent 16600 or 16700 Analyzer with Multi-ICE:

1. Install Multi-ICE on both the workstation you are debugging on and the workstation that the Multi-ICE interface unit connects to.

2. Configure the Multi-ICE server on the workstation that the Multi-ICE interface unit is connected to.

3. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7.

4. In the Code window, select File → Connection → Connect to Target... to display the Connection Control window.

5. Expand the top-level ARM-A-RR vehicle.

6. If Multi-ICE is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-1 on page B-2.
   c. If Multi-ICE is present, select the checkbox. If not, click Add DLL... and select Multi-ICE.dll from the Multi-ICE install directory. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close this dialog.

7. Right-click on the Multi-ICE entry and select Configure Device Info... from the context menu to show the ARM Multi-ICE Configuration dialog box (Figure B-2 on page B-3).

8. In the Connect tab, select the ARM processor that you are tracing from the list shown in the list of Available devices.
9. Display the Trace tab, as shown in Figure B-3 on page B-4.

10. Specify the ARM ADI driver in the Select a Trace Capture DLL... drop-down list. This specifies the driver file that is used to control the trace capture device and to carry out operations such as start and stop tracing:
   • If Gateway2.dll is present in the drop-down list, select it.
   • If Gateway2.dll is not present in the drop-down list, click Add and select Gateway2.dll from \bin. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.)

11. Click Configuration to display the Logic Analyzer Configuration dialog box. You must also select the appropriate startup option to indicate the level of initialization carried out by the debugger:
   • Select the Automatic option if you want the debugger to ensure, at start-up, that the logic analyzer is fully initialized to carry out tracing. In this case, you must specify the appropriate machine name, lister name, and configuration filename in the dialog box.
     You must specify the full directory path to the configuration file.
   • Select the Set-up Assistant option if you do not want the configuration file to be loaded by the debugger. The other parameters (lister name and machine name) are initialized with the given values. This mode is appropriate only when you are loading a configuration file from the logic analyzer user interface, which can be done using either of the following:
     — the Setup Assistant
     — the File Manager tool.

   If you are using the default logic analyzer configuration files provided by Agilent, you must:
   • set the machine name as ARM ETM Analyzer
   • set the lister name as ETM Data.

   ______ Note ______

   The default logic analyzer configuration files cannot be loaded directly by the analyzer. Instead, using the file manager tool on the analyzer user interface, you must load each file, save the configuration back to an appropriate filename, and specify this new name as the configuration to load.

12. Click OK to exit each of the dialog boxes successively.

13. Select ADI from the targets list in the Connection Control window, by expanding the second-level Multi-ICE entry, and selecting the processor connection, such as ARM966E-S.
For detailed information on setting up the Agilent 16600 or 16700 Logic Analyzer, see the Agilent documentation, which can be accessed from the website http://www.agilent.com. See Other publications on page x for details.
B.7 Agilent Trace Port Analyzer and Agilent Emulation Probe

This section describes how to set up RealView Debugger to connect to a combination of the Agilent Trace Port Analyzer and Emulation Probe. You must have already set up the logic analyzer software as part of the hardware setup (see Appendix B Setting up the Trace Software).

--- Note ---
ARM ADI is not supplied with RealView Developer Suite, and must be purchased separately.

---
To configure the Agilent Analyzer and probe:

1. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7.
2. Select File → Connection → Connect to Target... to display the Connection Control window.
3. Expand the top-level ARM_A_RR vehicle.
4. If ADI is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-1 on page B-2.
   c. If ADI is present, select the checkbox. If not, click Add DLL... and select Gateway2.dll from \bin. You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close the RDI Target List dialog.
5. Right-click on the ADI entry and select Configure Device Info... from the context menu to show the Gateway Configuration dialog box (Figure B-16 on page B-18).
6. Fill in the network address for the Emulation Probe, and click Lookup.
7. If your target is running in big-endian mode:
   a. Click Advanced to display the advanced configuration tab, as shown in Figure B-17 on page B-19.
   b. Click Big-endian.
8. Click Trace to display the trace configuration tab, as shown in Figure B-18 on page B-20.
Setting up the Trace Software

9. Specify the Gateway2 driver in the **Select a Trace Capture DLL...** drop-down list. This specifies the driver file that is used to control the trace capture device and to carry out operations such as start and stop tracing:
   - If *Gateway2.dll* is present in the drop-down list, select it.
   - If *Gateway2.dll* is not present in the drop-down list, click **Add...** and select *Gateway2.dll* from *\bin*. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.)

The configured dialog box is shown in Figure B-21.

![Gateway2 Configuration](image)

**Figure B-21 Configuring Gateway2**

10. Click **Trace Port Analyzer**.

11. Enter the network address of the Agilent Trace Port Analyzer in the **Network Address** field.

12. Click **OK** successively to exit each of the dialog boxes.

13. Select ADI from the targets list in the Connection Control window, by expanding the second-level **Multi-ICE** entry, and selecting the processor connection, such as **ARM966E-S**.
Note

For details on configuring for an Agilent Emulation Probe, see the Setting Up the Trace Port Analyzer chapter of the Trace Port Analysis for ARM ETM User's Guide.

For detailed information on setting up the Agilent Trace Port Analyzer and Agilent Emulation Probe, see the Agilent documentation, which can be accessed from the website http://www.agilent.com. See Other publications on page x for details.

For detailed information on setting up and using ARM ADI, see the ARM ADI User Guide.
B.8 Tektronix TLA 600 or TLA700 and ARM Multi-ICE

This section describes how to set up RealView Debugger to connect to a combination of ARM Multi-ICE and the Dragonfly Software Tektronix trace capture agent (not supplied by ARM Limited). Refer to the Multi-ICE Installation Guide or to the Multi-ICE User Guide for more information about Multi-ICE, and to the Tektronix TLA Logic Analyzer ARM ETM Support Package Instructions for more information on setting up the Tektronix interface software.

To install and set up the Tektronix TLA600 or TLA700 and Multi-ICE:

1. Configure the Multi-ICE server on the workstation that the Multi-ICE interface unit is connected to.

2. Select Start → Programs → ARM → RealView Developer Suite v2.1 → RealView Debugger v1.7.

3. Select File → Connection → Connect to Target... to display the Connection Control window.


5. If Multi-ICE is not present in the targets list, display it in the following way:
   a. Right-click on any target in the list to display a context menu.
   b. Select Add/Remove/Edit Devices... from the context menu to display the RDI Target List, shown in Figure B-1 on page B-2.
   c. If Multi-ICE is present, select the checkbox. If not, click Add DLL... and select Multi-Ice.dll from the Multi-ICE install directory. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close this dialog.

6. Right-click on the Multi-ICE entry and select Configure Device Info... from the context menu to show the ARM Multi-ICE Configuration dialog box (Figure B-2 on page B-3).

7. Depending on the location of the Multi-ICE server that you are using, click on either This computer or Another computer.
   If you select Another computer, in the subsequent dialog box type in the name of the remote workstation, or locate it using the tree control. Click on OK.

8. Select the ARM processor that you are tracing from the list shown in Available devices.
9. Optionally, enter your name in the **Connection name** text field. This is displayed in the Multi-ICE server window and can help if you are sharing the server with others.

10. Click on the **Trace** tab to make it visible (Figure B-22).
    
    The tab contains the controls required to configure the trace control software. The drop-down list labeled **Select a Trace Capture DLL...** contains the names of the currently available trace capture drivers. These drivers read the trace information from the ETM and translate it into the format required by RealView Debugger.

11. Select the Dragonfly driver *adst1a.dll* from the **Select a Trace Capture DLL...** drop-down list:
    
    - If *adst1a.dll* is present in the drop-down list, select it.
    - If *adst1a.dll* is not present in the drop-down list, click **Add...** and select *adst1a.dll* from the Dragonfly Software TLA installation directory. (You might have to configure Windows Explorer to ensure that files with the extension .dll are not hidden from view.)

    This specifies the driver file that is used to control the trace capture device and to carry out operations such as start and stop tracing.

---

**Figure B-22 Multi-ICE configuration dialog box showing the Trace tab**
12. Click on **Configure**... to display the Dragonfly trace capture configuration dialog box (Figure B-23).

![Figure B-23 Dragonfly TLA Configuration dialog box](image)

13. Configure the Dragonfly Software trace capture agent as described in *Tektronix TLA Logic Analyzer ARM ETM Support Package Instructions*.

14. Click **OK** successively to exit each of the dialog boxes.

15. Select Multi-ICE from the targets list in the Connection Control window, by expanding the second-level **Multi-ICE** entry, and selecting the processor connection, such as ARM966E-S.
B.9 Simulators using the Simulator Broker connection

This section describes how to connect to simulators that use the Simulator Broker connection, such as DSP Group targets.

To access the simulator:

1. Select File → Connection → Connect to Target... to display the Connection Control window.
2. Expand the entry Server Connection Broker.
3. Expand the entry localhost Simulator Broker.
4. Double-click on the required entry to start a simulator connection.
   The connection list expands to show your new connection. The Code window title bar is updated to show this connection.
5. Click File → Load Image... to load an executable file suitable for the simulator you are using.

By default, RealView Debugger is automatically configured with tracing enabled for ARM targets using preset values stored in the Logic_Analyzer settings group in the Advanced_Information block. However, to enable trace, you must ensure that Connect Analyzer/Analysis is selected in the Analysis window Edit menu. See Configuring trace options on page 2-48 for information on how to do this.
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 MultiTrace**
External collection unit for ARM Real-Time Trace.

**ARM state**
A processor that is executing ARM (32-bit) instructions is operating in ARM state.

*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  
Captive threads are all threads that can be brought under debugger control. Special threads, called non-captive threads, are essential to the operation of Running System Debug (RSD) and so are not under the control of RealView Debugger. Non-captive threads are grayed out in the GUI.

See also Running System Debug.

Complex tracepoint  
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.

See also Tracepoints.

Conditional breakpoint  
A breakpoint that halts execution when a particular condition becomes true. The condition normally references the values of program variables that are in scope at the breakpoint location.

Context menu  
See Pop-up menu.

Core module  
In the context of Integrator, an add-on development board that contains an ARM processor and local memory. Core modules can run stand-alone, or can be stacked onto Integrator motherboards.

See also Integrator

Current Program Status Register (CPSR)  
See Program Status Register.

DCC  
See Debug Communications Channel.

Debug Agent (DA)  
The Debug Agent resides on the target to provide target-side support for Running System Debug (RSD). The Debug Agent can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the debug communications channel (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state.

See also Running System Debug.

Debug Communications Channel (DCC)  
A debug communications channel enables data to be passed between RealView Debugger and the EmbeddedICE logic on the target using the JTAG interface, without stopping the program flow or entering debug state.

Debug With Arbitrary Record Format (DWARF)  
ARM code generation tools generate debug information in DWARF2 format.
Glossary

**Deprecated**
A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported 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 Macrocell (ETM)**
A block of logic, embedded in the hardware, that is connected to the address, data, and status signals of the processor. It broadcasts branch addresses, and data and status information in a compressed protocol through the trace port. It contains the resources used to trigger and filter the trace output.

**EmbeddedICE logic**
The EmbeddedICE logic is an on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

See also IEEE1149.1.

**Emulator**
In the context of target connection hardware, an emulator provides an interface to the pins of a real core (emulating the pins to the external world) and enables you to control or manipulate signals on those pins.

**Endpoint connection**
A debug target processor, normally accessed through an access-provider connection.

**ETM**
See Embedded Trace Macrocell.

**ETV**
See Extended Target Visibility.

**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 the user 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. Core modules are available that contain the processor and local memory.

Joint Test Action Group (JTAG)
An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a Joint Test Action Group (JTAG) interface port to communicate with processors. For further information refer to IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

JTAG
See Joint Test Action Group.
Glossary

**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.  
*See also* Big-endian.

**Multi-ICE**  
A JTAG-based tool for debugging embedded systems.

**Pop-up menu**  
Also known as *Context menu*. A menu that is displayed temporarily, offering items relevant to your current situation. Obtainable in most RealView Debugger windows or panes by right-clicking with the mouse pointer inside the window. In some windows the pop-up menu can vary according to the line the mouse pointer is on and the tabbed page that is currently selected.

**Processor core**  
The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

**Profiling**  
Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.

**Program Status Register (PSR)**  
*Program Status Register* (PSR), containing some information about the current execution context. It is also referred to as the *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR), which records information about an alternate processor mode.

**PSR**  
*See* Program Status Register.

**RDI**  
*See* 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.

Simple 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.

See also Tracepoints.

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 state
A processor that is executing Thumb (16-bit) instructions is operating in Thumb state.

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 need 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**

An unit within a complex tracepoint, similar to a simple tracepoint, which combines with other tracepoint units to create the complex 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.

**Vector Floating Point (VFP)**

A standard for floating-point coprocessors where several data values can be processed by a single instruction.

**VFP**

See 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.
Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

AAPCS 4-55
Absolute Time 2-86
Access provider nodes B-10, B-11, B-12
Access Type 2-87
Access type match, filter on 2-84
Access Type, sort by 2-89
Actions menu 4-45
Address as Symbol/Line 2-87
Address as value 2-87
Address column (Analysis window) 2-61
Address expression match, find 2-75
Address expression, match, filter on 2-81
Address mapping 2-55
Address, sort by 2-89
Address/Data 2-30
Agilent
  Emulation module 2-3
  Emulation Probe A-12, B-27
  E5903A A-12, A-13
Logic analyzer 2-3
Trace Port Analyzer 2-3, B-27
16600 or 16700 logic analyzer A-5, A-9, B-24
Analysis configuration, selecting 2-50
Analysis window 2-44
code window tracking 2-56
column types 2-60
Columns menu 2-86
Context menu 2-93
Find menu 2-72
Exit 2-92
Sort menu 2-89
Status bar 2-92
Status lines 2-64
Tabs 2-59
update 2-56
View menu 2-78
Analyzer properties, configuring 2-11, 2-49
AND Filters 2-85
Complex tracepoints Trace on X after Y 2-37
Dialog
  Trace if X after Y 2-37
  Trace on X after Y 2-37
Append new buffer to Existing 2-58
ARM CPU macrocell 2-8
ARM Multi-ICE 2.0 configuration dialog box B-3, B-7, B-15, B-24, B-30
ARM processors 2-5
ETM 2-8
ARM-ARM-PP connection 3-8
ASIC 2-8, 2-19
Asterisk, beside connection 5-16
Attach thread 4-39
Attach Window to a Connection 5-23
Attachment
  Color Box 5-25
  inheriting 5-25
  of board 5-21
  of window 5-8, 5-21, 5-25
  status 4-27

ARM DUI 0174E Copyright © 2002-2004 ARM Limited. All rights reserved. Index-1
Automatic Update on Append to buffer  2-58
Automatic Update on New buffer   2-58
Auto-range   2-76, 2-82
AXYS Design Automation Inc   1-4, 3-2, 3-4

B

Board
attachment   5-21
Break trigger group   4-47
changing   4-50
disappeared   4-47
empty   4-47
BREAKINSTRUCTION command   4-59
Breakpoints
and memory mapping   4-48
break trigger group   4-47
break trigger group changing   4-50
break trigger group disappears   4-47
break trigger group empty   4-47
classes   4-52
colors   4-49, 4-53
conditional in RSD   4-47
hardware in RSD   4-47
HSD   4-46
in the Break/Tracepoints pane   4-53
in the Set Address/Data
Break/Tracepoint dialog   4-52
in the Set Address/Data
Break/Tracepoint dialog box   4-53
RSD   4-46
RTOS support   4-46, 4-48, 4-52, 4-53
setting   4-48
Broadcast data links   5-37
Buffer, clear   2-56
Buffer, load from file
Trace buffer, load from file   2-45, 2-90
Buffer, Save to file
Trace buffer, Save to file   2-90
B=>E% column (Analysis window)   2-62, 2-87
Call-graph   2-70
Captive thread   4-31, 4-45
Capturing profiling information   2-100
CARMETM   A-8, A-10
Changing the trace display   2-86
Children of function   2-70
Clear Trace Buffer   2-56
Clearing tracepoints   2-42
Close Loaded File   2-91
Cmd tab
in Output pane   5-17
Code view   2-59
Code window
attaching to multiple connections   5-23
attachment   4-27, 5-18
Color Box   5-18
compared to connection   5-5
creating   5-16
color box   5-17
Code Window Tracking   2-56
Coherency
in multiprocessor mode   5-2, 5-37
Collect data Around Trigger   2-52, 2-53
Collect data Before Trigger   2-52, 2-53
Color Box   5-18, 5-23
changing windows attachment   5-25
Column types (Analysis window)   2-60
Columns menu (Analysis window)   2-86
Commands
ADDFILE   4-22
aos_resource-list   4-64
BREAKINSTRUCTION   4-59
dox_resource-list   4-63
HALT   4-61
Imstat   3-7
OCTRL   4-58
RELOAD   4-22
RTOS support   4-58
STOP   4-62
Comments from users   xii
Communication guarding   5-50
Complex tracepoints   2-2, 2-22, 2-29
address/data   2-30
Trace on X after Y executed N times   2-38
Concepts
Code window   5-4
connection state   5-5
cross-triggering   5-47
debug target   5-5
debugger core   5-5
DSP   3-2
loose synchronization   5-47
multiprocessor mode   5-2
processor group   5-45
RTOS   4-2
skid   5-45
SMP   5-37
synchronization   5-45
target debug interface   5-5
target hardware   5-5
tight synchronization   5-47
Configuration settings
ordering   4-13, 4-21
precedence   4-13, 4-21
Configure
analyzer properties   2-49
Configure ETM
Cycle accurate tracing   2-18
Enable Timestamps   2-18
Memory map decode   2-19
Configure ETM dialog   2-13
Configure the ETM   2-9
Configure trace capture   2-9, 2-10, 2-21
Configuring
Agilent 16600 or 16700 logic analyzer   A-7, B-17
Analyzer properties   2-11
trace options   2-48, 2-56
Connect
menu   5-15, 5-20
Connect Analyzer/Analysis   2-48
Connect mode
defining   5-15
not honored   4-19
substitution   4-19
Connect tab
Connection Control window   5-14
Connecting
effects on projects   5-33
setting connect mode   5-15
Connecting to a running target
RTOS 4-16, 4-23
Connecting to a target 2-9
Connecting to DSP 3-2
Connection
active connections list 5-25
active list 5-16
asterisk by 5-16, 5-21
attach windows 5-20
button 5-20
changing 5-15
changing current 5-22
changing the debug view 5-25
compared to Code window 5-5
creating new RTOS-enabled connection 4-8
current 5-8, 5-16, 5-21
displaying information 5-19
in Code window title bar 5-17
selecting a new connection 5-20
state information 5-5
Connection Broker
local host connection 3-4
shut down 3-6
Connection button 5-15, 5-20
Connection Control window 5-13
Connect tab 5-14
Synch tab 5-14, 5-51, 5-52, 5-53
Connection Properties window 4-12, B-14
Connections list 5-20
Continue Collecting on Buffer Full 2-53
Copy 2-48, 2-94
Count column (Analysis window) 2-61, 2-63
Count of hits 2-87
Count, sort by 2-89
Cross-triggering 5-47, 5-50
controls 5-53
hardware example 5-48
software example 5-48
synchronization 5-50, 5-51
Current connection 5-7, 5-16, 5-22
changing 5-22
Current thread 4-28, 4-30
changing in CLI 4-28
CLI commands 4-28, 4-31
details in Output pane 4-28
in RSD 4-28, 4-31
in unattached window 4-30
using the Thread button 4-29
Cycle-accurate tracing 2-18
Cycles 2-57
D
DA
Debug Agent 1-3, 4-4, 4-16
Data abort example (trace) 2-96
Multi-ICE 2-96
RealView ICE 2-96
Data synchronization lost 2-64
Data Value in Decimal 2-87
Data Value in Hex 2-87
Data value match, filter on 2-82
Data value match, find 2-76
Data value, sort by 2-89
Data view 2-59
Data, interpreting 2-58
DCC 4-5
Deadlines, scheduling 4-2
Debug Agent 1-3, 4-4, 4-16
DCC 4-5
disconnect on exit 4-25
ICTC 4-5
ICTM 4-31, 4-39
in thread list 4-31
undefined exceptions catch 4-17
viewing status 4-37
Debug communications channel 4-5
Debug interface unit 5-8, 5-11, 5-34, 5-35, 5-36, 5-44
Debug State (trace status line) 2-66
Debug target concept 5-5
disconnecting 5-26
viewing different targets 5-3, 5-12
Default Ring-buffered last N traces 2-50
Define Processor Speed for Scaling 2-58
De-Multiplexed 2-14
Desktop
attaching windows 5-18
Color Box 5-18
Dialog 2-40
Configure ETM 2-13
Information 2-94
List Selection 2-25, 2-57, 2-84
Prompt 2-52, 2-58, 2-72, 2-75, 2-77, 2-79, 2-80, 2-81, 2-82, 2-83, 2-85
Select Trace file to Save to 2-91
Trace if X after A==B 2-39
Trace if X after ‘n’ executions of Y 2-38
Dialog boxes
ARM ADI configuration B-18
ARM Multi-ICE configuration B-3, B-7
TLA configuration B-32
Direct connect, Multi-ICE 3-7, 3-8
Disabling tracepoints 2-41
Disassembly tab 2-21
Disconnect
from targets 5-25
menu 5-26
Disconnect All 5-29
Disconnect mode
defining 5-28
Disconnecting 5-25, 5-26
active connection 5-28
all 5-29
all connections 5-29
by exiting 5-28
current connection 5-27
effects 5-26
effects on projects 5-33
from target 5-26
last connection 5-27
setting disconnect mode 5-28
using Connection Control window 5-27
Disconnection options
RTOS support 4-24
Displaying
Resource Viewer window 5-24
thread information 4-44
Downloads
RTOS Awareness 1-3, 4-7
Dom view 2-59
DSP 3-2
color 3-2
connecting to hardware 3-7
Index

connecting with RealView ICE | 3-7
licensing | 3-3
local host connection | 3-2, 3-4
Multi-ICE direct connect | 3-7
MS6621 | 3-2
Oak | 3-2
simulator | 1-4, 3-2, 3-4
support | 3-2
TeakDSP | 3-2
TeakLite | 3-2
with multiple targets | 3-7

F

Feedback | xii
FIFO buffer | 2-4, 2-16
FIFO highwater | 2-16, 2-17
FIFO overflow | 2-19, 2-64
FIFOFULL logic | 2-13
File Editor window | 2-21
File menu
  in Resource Viewer window | 4-41
Files
  .bcd for RTOS | 4-7, 4-12
  .brd for RTOS | 4-14
  .dll for RTOS | 4-7
  .dll, showing | B-25, B-28
Filter on Access Type Match | 2-84
Filter on Address Expression Match | 2-81
Filter on Data Value Match | 2-82
Filter on Percent Time Match | 2-85
Filter on Position Match | 2-78
Filter on Raw Address Match | 2-80
Filter on Symbol Name Match | 2-83
Filter on Time Match | 2-79
Filtering captured information | 2-78
Filtering trace | 2-21
  AND filters | 2-85
  filter on access type match | 2-84
  filter on address expression match | 2-81
  filter on data value match | 2-82
  filter on percent time match | 2-85
  filter on position match | 2-78
  filter on raw address match | 2-80
  filter on symbol name match | 2-83
  filter on time match | 2-79
  save filtered buffer to file | 2-91
Find Address Expression Match | 2-75
Find corresponding source code | 2-42
Find Data Value Match | 2-76
Find menu | 2-72
Find menu (Analysis window) | 2-72
Find next | 2-94
Find Position Match | 2-73
Find previous | 2-94
Find Raw Address Match | 2-74
Find Symbol Name Match | 2-77
Find Time Match | 2-73
Find Trigger | 2-72
Func view | 2-59

G

Gateway configuration dialog box | B-17, B-27
General Purpose Input/Output. See GPIO.
Getting started with trace | 2-7
GPIO | 2-13
Group Name/Type selector dialog | 4-9

H

Half-rate clocking | A-7, A-10, B-22
HALT command | 4-61
Halted System Debug
  HSD | 1-3, 4-4
  Hard real-time | 4-2
Hardware
  platforms | 1-8
  requirements | 2-3
  resources | 2-4
Hardware solution for trace | 2-7
Histogram column (Analysis window) | 2-63
Histogram View | 2-69
Histogram, logarithmic scale | 2-71
HSD
  breakpoints | 4-46
  disable on exit | 4-25
Halted System Debug | 1-3, 4-4
  switching to RSD | 1-3, 4-4, 4-37, 4-42, 4-58, 4-61, 4-62

I

ICTC | 4-5
ICTM | 4-31, 4-39
IMP Comms Target Controller | 4-5
IMP Comms Target Manager | 4-31, 4-39
Incorrect synchronization address | 2-65
Information about threads | 4-44
Information dialog | 2-94
Interpretation of Data | 2-87
Index

Interpreting trace data   2-58
Interrupts
  when loading to an RTOS   4-25
IP address   B-21, B-27, B-28
IRQ
  when loading to an RTOS   4-25

J
JTAG
  interface unit   2-3, 2-9, 3-7
  scan path   5-13
  specification   x

L
Last N traces   2-50
Licenses   1-6, 5-15
Linear scale   2-71
List selection dialog   2-25
Load Trace Buffer from File   2-45, 2-90
Loading
  an image   2-9, 2-21
  trace information   2-90
Logarithmic scale   2-71
Logging commands and output   5-17
Logic Analysis System   A-6, A-7, A-10
Logic analyzer   B-24
  Agilent   2-4
  automatic configuration   B-25
Logic Analyzer Configuration dialog box   B-21, B-25
Logic analyzers   B-21, B-28
  Agilent   2-3
  Agilent 16600 or 16700   A-5, A-9
  automatic configuration   B-22
MultiTrace   A-2
RealView Trace   A-4
Setup Assistant   B-22, B-25
TLA 600   A-15
Loose synchronization   5-47

M
MaxSim simulator   1-4, 3-2, 3-4
MCR   2-17
Memory
  and breakpoints   4-48
Memory map decode   2-19
Memory pane   2-21
Micro-Seconds   2-57
Mictor connector   A-9, A-12
Milli-Seconds   2-57
Min/Max B->E column (Analysis window)   2-63, 2-88
Modifying
  local variables   4-56
  shared variables   4-56
  tracepoints   2-41
MRC   2-17
Multi-ICE   2-3, A-2, A-9, B-2, B-6, B-15, B-24
  debug interface   5-5
  with multiple targets   5-35, 5-36
Multi-ICE direct connect   3-7, 3-8
Multiplexed   2-14
Multiprocessor
  debugging   1-5
  mode
    active connections   5-15
    active project   5-30
    architecture   5-3, 5-9, 5-12
    architecture overview   5-2
    Code window   5-4
    Coherency   5-2, 5-37
    communication guarding   5-50
    concept   5-2
    Connect tab   5-14
    connecting to a target   5-13
    Connection button   5-20
    connection state   5-5
    Connections list   5-20
    cross-triggering   5-47, 5-50
    current connection   5-15
    debugger core   5-5
    first connection   5-4
    license   5-2
    loose synchronization   5-47
    no license   5-15
    project binding   5-30
    Project Control dialog box   5-31
    project environment   5-30
    projects   5-29
    RTOS architecture   5-11
    second connection   5-6
Synch tab   5-14, 5-51
  synchronization   5-2, 5-13, 5-45
  synchronization and cross-triggering   5-50, 5-51
  synchronization facilities   5-51
  synchronization group   5-45
  synchronization skid   5-45
  synchronization skid example   5-46
  synchronization terms   5-45, 5-47
  synchronizing processors   5-53
  target connections   5-2
  tight synchronization   5-47
MultiTrace   2-3, 2-16, A-2, B-2, B-6

N
Name, sort by   2-89
Nano-Seconds   2-57
Non-captive thread   4-31
Non-ETM processors   2-6
Non-intrusive   2-2
Normal   2-14

O
Oak
  simulator   1-4, 3-2, 3-4
OS Abstraction DLL   1-3
OS Awareness   1-3
OS marker
  context menu   4-37
  error status   4-37
  HSD in the Process Control pane   4-35
  in the Process Control pane   4-34, 4-35
  RSD in the Process Control pane   4-35
  status and meaning   4-36
  OSCTRL command   4-58
  Other column (Analysis window)   2-61
Output pane
  Cmd tab   5-17
Index

P
Parents of function 2-70
Parent/Child % ages Relative to Whole Time 2-72
Percent time match, filter on 2-85
Physical to Logical Address Mapping 2-55
Pico-Seconds 2-57
Plugins
for RTOS 4-7
Point-to-point data links 5-37
Position 2-86
Position match
filter on 2-78
find 2-73
Print Trace Lines 2-92
Process
and threads 4-3
in Process Control pane 4-34
Process Control pane 4-34, 4-37
Process tab 4-34
Thread tab 4-37
Processor group
in multiprocessor synchronization 5-45
Processor stalling
FIFO highwater 2-16, 2-17
Profile view 2-59
Program execution 2-10, 2-43
Project Control dialog box
in multiprocessor mode 5-31
Projects
and windows attachment 5-31
binding in multiprocessor mode 5-30, 5-32
bound 5-18
default binding 5-32
effects of connecting 5-33
effects of disconnecting 5-33
in multiprocessor mode 5-30
working with multiple connections 5-30
working with multiple windows 5-31
Prompt dialog 2-75, 2-83

R
Raw address match
find 2-74
Raw address, match, filter on 2-80
Raw view 2-59
Real-time 2-2
RealView ARMulator ISS 2-4, 2-6
RealView Debugger
documentation suite ix
extensions 1-2
single processor license 5-15
tracing 2-2
windows attachment 5-18
RealView ICE
Disconnect mode 4-19
RealView ICE 2-3, A-4, B-10
adding targets 4-9
configuring new connection 4-8
Connect mode 4-19
debg interface 5-5
troubleshooting connections 5-35
with multiple targets 5-34
RealView Trace 2-3, 2-16, A-4, B-10
Registers
viewing for threads 4-55
viewing with RealView ICE 5-34
Relative Time 2-86
sort by 2-89
Remote configuration B-25, B-27
Resource sharing 5-37
Resource Viewer window 4-40, 4-44,
5-18, 5-24
Action menu 4-44
Action menu for captive threads 4-45
Action menu parameters 4-44
action parameters 4-44
attachment status 5-20, 5-25
changing connections 5-19
Conn tab 4-40, 5-18
Details area 4-44
File menu 4-41
Resources list 4-40, 4-44
Resources toolbar 4-43
RSD menu 4-42
RTOS plugin 4-44
RTOS tabs 5-19
tabs in 4-40, 4-44
View menu 4-41
Resources 2-4
Resources toolbar
in Resource Viewer window 4-43
Reverse Sort 2-89, 2-90
Ring-buffered last N traces 2-50
RSD
breakpoints 4-17, 4-46
contingual breakpoints 4-47
disable configuration setting 4-16
disable on exit 4-25
enable configuration setting 4-16
hardware breakpoints 4-47
Running System Debug 1-3, 4-4
switching to HSD 1-3, 4-4, 4-37,
4-42, 4-58, 4-61, 4-62
system breakpoints 4-46
thread breakpoints 4-47
RSD menu
in Resource Viewer window 4-42
RTOS 4-1, 4-2
actions on objects 4-44
Advanced_Information block 4-14
ARM_config configuration settings 4-17
Base_address configuration setting 4-15
Board/Chip definition files 4-7
break trigger group 4-47
Breakpoint Classes 4-52
breakpoints 4-46, 4-48, 4-49, 4-52,
4-53
Break/Tracepoints pane 4-53
CLI commands 4-58
concept 4-2
conditional breakpoints 4-47
configuration settings 4-14, 4-24
configuration settings conflicts 4-16
configuring an RTOS-enabled
connection 4-12
Connect mode configuration rules 4-18, 4-19
connecting to a running target 4-23
connecting to a target 4-22
Connect_mode configuration
settings 4-18, 4-19, 4-23
creating a new RTOS-enabled
connection 4-8
current thread 4-28
<table>
<thead>
<tr>
<th><strong>Index</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DCC</strong> 4-5</td>
</tr>
<tr>
<td><strong>Debug Agent</strong> 4-4, 4-16</td>
</tr>
<tr>
<td><strong>Disconnect mode configuration rules</strong> 4-18, 4-19</td>
</tr>
<tr>
<td><strong>disconnection configuration settings</strong> 4-24</td>
</tr>
<tr>
<td><strong>disconnection options</strong> 4-24</td>
</tr>
<tr>
<td><strong>Disconnect_mode configuration settings</strong> 4-18, 4-19, 4-24</td>
</tr>
<tr>
<td><strong>Exit_Options configuration setting</strong> 4-15, 4-16, 4-24</td>
</tr>
<tr>
<td><strong>Exit_Options prompt</strong> 4-24</td>
</tr>
<tr>
<td><strong>formatting memory</strong> 4-56</td>
</tr>
<tr>
<td><strong>hardware breakpoints</strong> 4-47</td>
</tr>
<tr>
<td><strong>HSD</strong> 1-3, 4-4</td>
</tr>
<tr>
<td><strong>HSD breakpoints</strong> 4-46</td>
</tr>
<tr>
<td><strong>ICTC</strong> 4-5</td>
</tr>
<tr>
<td><strong>ICTM</strong> 4-31, 4-39</td>
</tr>
<tr>
<td><strong>interrupts when loading</strong> 4-25</td>
</tr>
<tr>
<td><strong>IRQ</strong> 4-25</td>
</tr>
<tr>
<td><strong>jobs</strong> 4-2</td>
</tr>
<tr>
<td><strong>loading images</strong> 4-25, 4-26</td>
</tr>
<tr>
<td><strong>loading the plugin</strong> 4-15</td>
</tr>
<tr>
<td><strong>Load_when configuration setting</strong> 4-15, 4-16</td>
</tr>
<tr>
<td><strong>Map tab</strong> 4-34</td>
</tr>
<tr>
<td><strong>multiprocessor architecture</strong> 5-11</td>
</tr>
<tr>
<td><strong>objects</strong> 4-44</td>
</tr>
<tr>
<td><strong>OS Awareness Downloads</strong> 1-3</td>
</tr>
<tr>
<td><strong>OS marker</strong> 4-34, 4-35</td>
</tr>
<tr>
<td><strong>OS state</strong> 4-25</td>
</tr>
<tr>
<td><strong>plugins</strong> 1-3, 4-7</td>
</tr>
<tr>
<td><strong>Process tab</strong> 4-34</td>
</tr>
<tr>
<td><strong>Real-time applications</strong> 4-2</td>
</tr>
<tr>
<td><strong>resetting OS state when reloading</strong> 4-25</td>
</tr>
<tr>
<td><strong>Resource Viewer window</strong> 4-40, 4-41, 4-43</td>
</tr>
<tr>
<td><strong>RSD</strong> 1-3, 4-4</td>
</tr>
<tr>
<td><strong>RSD breakpoints</strong> 4-46</td>
</tr>
<tr>
<td><strong>RSD configuration setting</strong> 4-16</td>
</tr>
<tr>
<td><strong>RTOS Awareness Downloads</strong> 1-3, 4-7</td>
</tr>
<tr>
<td><strong>RTOS configuration settings</strong> 4-14</td>
</tr>
<tr>
<td><strong>Set Address/Data Break/Tracepoint dialog box</strong> 4-52, 4-53</td>
</tr>
<tr>
<td><strong>special threads in the Thread tab</strong> 4-39</td>
</tr>
<tr>
<td><strong>data structures</strong> 4-40</td>
</tr>
<tr>
<td><strong>specifying image target name</strong> 4-26</td>
</tr>
<tr>
<td><strong>stepping threads</strong> 4-26</td>
</tr>
<tr>
<td><strong>System_Stop configuration setting</strong> 4-16</td>
</tr>
<tr>
<td><strong>Thread button</strong> 4-28</td>
</tr>
<tr>
<td><strong>Thread list</strong> 4-29</td>
</tr>
<tr>
<td><strong>Thread menu</strong> 4-33</td>
</tr>
<tr>
<td><strong>Thread tab</strong> 4-34, 4-37</td>
</tr>
<tr>
<td><strong>threads in the Thread tab</strong> 4-38</td>
</tr>
<tr>
<td><strong>Vendor configuration setting</strong> 4-15</td>
</tr>
<tr>
<td><strong>viewing registers</strong> 4-55</td>
</tr>
<tr>
<td><strong>RTOS Awareness</strong> 1-3, 4-7</td>
</tr>
<tr>
<td><strong>Running System Debug</strong> RSD 1-3, 4-4</td>
</tr>
<tr>
<td><strong>RVConfig dialog box</strong> B-12</td>
</tr>
<tr>
<td><strong>S</strong></td>
</tr>
<tr>
<td><strong>Save Filtered Trace Buffer to File</strong> 2-91</td>
</tr>
<tr>
<td><strong>Save Trace Buffer to File</strong> 2-90</td>
</tr>
<tr>
<td><strong>Saving trace information</strong> 2-90</td>
</tr>
<tr>
<td><strong>Scale Time Units</strong> 2-57</td>
</tr>
<tr>
<td><strong>Scale, linear</strong> 2-71</td>
</tr>
<tr>
<td><strong>Scale, logarithmic</strong> 2-71</td>
</tr>
<tr>
<td><strong>Scaling</strong> defining the processor speed 2-58</td>
</tr>
<tr>
<td><strong>Scheduling to deadlines</strong> 4-2</td>
</tr>
<tr>
<td><strong>Seconds</strong> 2-57</td>
</tr>
<tr>
<td><strong>Select Analysis Configuration</strong> 2-50</td>
</tr>
<tr>
<td><strong>Set Amount of Trace buffer to read</strong> 2-51</td>
</tr>
<tr>
<td><strong>Set Trace Buffer Size</strong> 2-50</td>
</tr>
<tr>
<td><strong>Set Trigger</strong> 2-25</td>
</tr>
<tr>
<td><strong>Setting tracepoints</strong> complex 2-105</td>
</tr>
<tr>
<td><strong>Setup Assistant, in logic analyzer</strong> B-22, B-25</td>
</tr>
<tr>
<td><strong>Set/Edit Event Triggers</strong> 2-55</td>
</tr>
<tr>
<td><strong>Set/Toggle Trace Point...</strong> 2-25</td>
</tr>
<tr>
<td><strong>Set Trigger</strong> 2-25</td>
</tr>
<tr>
<td><strong>Trace End Point</strong> 2-27</td>
</tr>
<tr>
<td><strong>Shared memory, as comms link</strong> 5-37</td>
</tr>
<tr>
<td><strong>Sharing multiprocessor resources</strong> 5-37</td>
</tr>
<tr>
<td><strong>Show code</strong> 2-42</td>
</tr>
<tr>
<td><strong>Show Details View</strong> 2-57</td>
</tr>
<tr>
<td><strong>Show Position relative to Trigger</strong> 2-57</td>
</tr>
<tr>
<td><strong>Simple breakpoints</strong> setting using Set/ToggleTrace Point 2-23</td>
</tr>
<tr>
<td><strong>Simple trace points</strong> Set/Toggle Trace Point... 2-25</td>
</tr>
<tr>
<td><strong>Simple tracepoints</strong> 2-2, 2-22</td>
</tr>
<tr>
<td><strong>Simulator</strong> 2-4</td>
</tr>
<tr>
<td><strong>Skid</strong> 5-45, 5-50</td>
</tr>
<tr>
<td><strong>example</strong> 5-46</td>
</tr>
<tr>
<td><strong>SMP</strong> 5-37</td>
</tr>
<tr>
<td><strong>Soft real-time</strong> 4-2</td>
</tr>
<tr>
<td><strong>Source tab</strong> 2-21</td>
</tr>
<tr>
<td><strong>Start of Excluded Trace Range</strong> 2-28</td>
</tr>
<tr>
<td><strong>Start of Trace Range</strong> 2-27</td>
</tr>
<tr>
<td><strong>Starting trace capture</strong> 2-42</td>
</tr>
<tr>
<td><strong>Status bar (Analysis window)</strong> 2-92</td>
</tr>
<tr>
<td><strong>Status lines</strong> 2-23, 2-64</td>
</tr>
<tr>
<td><strong>STOP command</strong> 4-62</td>
</tr>
<tr>
<td><strong>Stop processor on buffer full</strong> 2-52</td>
</tr>
<tr>
<td><strong>Stop processor on trigger</strong> 2-53</td>
</tr>
<tr>
<td><strong>Stop-Collect-Run profiling (intrusive)</strong> 2-50</td>
</tr>
<tr>
<td><strong>Store Control-Flow Changes Only</strong> 2-52</td>
</tr>
<tr>
<td><strong>Support for RTOS</strong> 1-3</td>
</tr>
<tr>
<td><strong>Supported hardware</strong> 1-8</td>
</tr>
<tr>
<td><strong>Suppress data on FIFO full</strong> 2-19</td>
</tr>
<tr>
<td><strong>Symbol name match</strong> filter on 2-83</td>
</tr>
<tr>
<td><strong>find</strong> 2-77</td>
</tr>
<tr>
<td><strong>Symbolic column (Analysis window)</strong> 2-61, 2-62</td>
</tr>
<tr>
<td><strong>Symmetric Multiprocessor</strong> 5-37</td>
</tr>
<tr>
<td><strong>Synch tab</strong> 5-51, 5-52</td>
</tr>
<tr>
<td><strong>Connection Control window</strong> 5-14</td>
</tr>
<tr>
<td><strong>cross-triggering controls</strong> 5-53</td>
</tr>
<tr>
<td><strong>Synchronization</strong> communication guarding 5-50</td>
</tr>
<tr>
<td><strong>concept</strong> 5-45</td>
</tr>
<tr>
<td><strong>cross-triggering</strong> 5-47, 5-50, 5-51</td>
</tr>
<tr>
<td><strong>cross-triggering controls</strong> 5-53</td>
</tr>
<tr>
<td><strong>cross-triggering example</strong> 5-48</td>
</tr>
<tr>
<td><strong>cross-triggering hardware example</strong> 5-48</td>
</tr>
<tr>
<td><strong>example</strong> 5-53</td>
</tr>
<tr>
<td><strong>Execution controls</strong> 5-52</td>
</tr>
<tr>
<td><strong>facilities</strong> 5-51</td>
</tr>
</tbody>
</table>
loose  5-47
skid  5-45
skid example  5-46
terms  5-45, 5-47
tight  5-47
using RealView ICE  5-34
Synchronization group  5-45
Synchronization lost  2-64
Synchronization Lost (trace status line)  2-64
Synchronization point  2-64
Synchronized on Run  5-52
Synchronized on Step  5-51, 5-52
Synchronized on Stop  5-52
System breakpoints
   RSD  4-46

T
Tabbed view types  2-59
Target connection  2-9
Target debug interface  5-5, 5-8, 5-11, 5-34, 5-35, 5-36, 5-44
Target hardware  5-5
Target nodes  B-11
Target vehicle nodes  B-10
Task Control Block
   TCB  4-55
TCB
   used by threads  4-55
Tektronix  B-30
   TLA 600 and 700  A-15
Tektronix logic analyzer  2-4
Terminology  Glossary-1
Thread
   and processes  4-3
   attached windows  4-27, 5-21
   attaching to windows  4-31
   attachment  4-39
   button  4-28
   captive  4-31
   changing the current thread  4-29
   changing variables  4-56
   Color Box  4-29
   current  4-28, 4-30
   details of  4-30
   displaying the thread list  4-29
   File menu  4-33
grayed in thread list  4-31
HSD  4-4
   in attached window  4-30
   in the thread list  4-29
   list  4-29
   markers  4-38
   non-captive  4-31
   not controlled by RealView
      Debugger  4-31
   registers  4-55
   Resource Viewer window  4-40
   RSD  4-4
   selecting a new thread  4-29
   selection box  4-32
   stepping  4-53
   stepping in attached window  4-55
   stepping in background  4-55
   stepping in unattached window  4-54
   unattached windows  4-27
   updating memory views  4-56
   updating views  4-56
   updating watches  4-56
   viewing in Code window  4-30
Thread breakpoints
   RSD  4-47
   stepping  4-53
   Thread button  4-28
   unavailable  4-35
   Thread list  4-29
   cycling  4-29
   Thread tab  4-37
   Debug Agent  4-39
   special threads  4-39
   threads view  4-38
   unavailable  4-35
   Threads
      from the File menu  4-33
      HSD  1-3
      see also  Thread
Tight synchronization  5-47
Time match
   filter on  2-79
   find  2-73
Time Measure from Selected  2-94
Time Measure from Triggered  2-94
Timestamps  2-18
Time/cycl column (Analysis window)  2-60
Time/Entry, sort by  2-89
Timing specifications, target  A-1
TLA 600 and 700, Tektronix  A-15
Trace
   address only  2-54
   capture hardware  2-9
   changing the display  2-86
   configuring capture  2-10
   control method  B-20, B-25, B-28
   data and address  2-54
   data only  2-54
   data width  B-22
   examples  2-95
   finding information  2-72
   getting started  2-7
   hardware solution  2-7
   interpreting the data  2-58
   overview  1-2
   using the examples  2-7
Trace buffer, clear  2-56
Trace buffer, save to file  2-91
Trace capture
   configuring  2-21
   starting  2-42
Trace capture hardware  2-3
Trace Data Value Access  2-31
Trace Data Value Read  2-30
Trace Data Value Write  2-31
Trace End Point  2-27
Trace FIFO Overflow (trace status line)  2-64
Trace hardware requirements  2-3
Trace if X after Y dialog box  2-37
Trace if X after ‘n’ executions of Y
   dialog box  2-38
Trace Instr Exec  2-30
Trace Instr Fetch  2-30
Trace on X after Y executed N times  2-38
Trace options
   configuring  2-48, 2-56
Trace Pause  2-23
Trace Pause (trace status line)  2-66
Trace Port Analyzer  2-3
Trace Port Analyzer, Agilent  B-21, B-28
Trace port mode  2-14
Tracepoints
   clearing  2-42
complex 2-2, 2-22, 2-29, 2-105
  disabling 2-41
  editing 2-40
  modifying 2-41
  simple 2-2, 2-22
Trace, filtering 2-21
Tracing 2-2
  coprocessors 2-17
  examples 2-95
  procedure 2-9
  saving and loading information 2-90
Tracing Enabled 2-49
Tracing fast targets A-1
Track in Code Window 2-46, 2-93
Trigger controls 5-53
Trigger, find 2-72
Troubleshooting
  RealView ICE connections 5-35

X

XScale B-15
XScale processor 2-8

Numerics

1st column (Analysis window) 2-62

Symbols

*, beside connection 5-16
+Time column (Analysis window) 2-60

U

Update 2-56
Use Logarithmic Scale for Histogram 2-71
User’s comments xii

V

Variables, changing in threads 4-56
View menu
  in Resource Viewer window 4-41
View menu (Analysis window) 2-78

W

Windows
  attachment 5-18
  Build-Tool Properties 4-8
  Color Box 5-18
  Resource Viewer 4-40
Windows attachment 5-8, 5-21, 5-25