RealView Debugger Extensions
User Guide

Copyright © 2002, 2003 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 1.5 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 2.0.</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 ................................................................................... xiii

#### Chapter 1  Introduction to RealView Debugger Extensions
1.1 About RealView Debugger extensions ................................................. 1-2
1.2 Licensing .................................................................................. 1-5
1.3 Supported platforms ...................................................................... 1-6
1.4 Supported hardware ...................................................................... 1-7

#### Chapter 2  Tracing with RealView Debugger
2.1 About tracing with RealView Debugger ................................................ 2-2
2.2 Getting started ............................................................................. 2-8
2.3 Configuring the ETM .................................................................... 2-12
2.4 Configuring trace capture ................................................................. 2-18
2.5 Using the Analysis window ............................................................... 2-39
2.6 Examples of using trace in RealView Debugger ................................. 2-85

#### Chapter 3  RTOS Support
3.1 About Real Time Operating Systems (RTOSs) ................................. 3-2
3.2 Using RealView Debugger RTOS support ....................................... 3-4
3.3 Associating threads with views ....................................................... 3-8
## Contents

3.4 Using the Resource Viewer window ......................................................... 3-12
3.5 Working with threads in the Process Control pane .............................. 3-14
3.6 RTOS resource CLI commands ............................................................... 3-15

### Chapter 4  DSP Support
4.1 About DSPs and RealView Debugger DSP support ............................ 4-2
4.2 Using the DSP ..................................................................................... 4-3

### 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 ................................................................. 5-10
5.4 Display coherency .................................................................................. 5-30
5.5 Processor execution synchronization ..................................................... 5-38

### Appendix A  Setting up the Trace Hardware
A.1 ARM MultiTrace and ARM Multi-ICE .................................................. A-2
A.2 Agilent 16600 or 16700 logic analyzer and Emulation Probe ................ A-4
A.3 Agilent 16600 or 16700 logic analyzer and Multi-ICE ............................ A-8
A.4 Agilent Emulation Probe and Trace Port Analyzer (E5904B) .............. A-11
A.5 Tektronix TLA 600 or TLA 700 logic analyzer and Multi-ICE .............. A-14

### Appendix B  Setting up the Trace Software
B.1 ARM MultiTrace and ARM Multi-ICE .................................................. B-2
B.2 ARM Multi-ICE for XScale ................................................................. B-6
B.3 Agilent 16600 or 16700 Logic Analyzer and Emulation Probe ................ B-8
B.4 Agilent 16600 or 16700 Logic Analyzer and ARM Multi-ICE ................ B-15
B.5 Agilent Trace Port Analyzer and Agilent Emulation Probe .................. B-18
B.6 Tektronix TLA 600 or TLA700 and ARM Multi-ICE ............................. B-21
B.7 ARMulator ETM emulation ................................................................. B-24
B.8 Simulators using the Simulator Broker connection .............................. B-26

Glossary
Preface

This preface introduces the *RealView™ Debugger Extensions v1.6 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, available by license only.
- 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 licensees 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, depending on the feature they have licensed.

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.

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

**Chapter 4 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 either running on multiple processors or using multiple threads.

**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.
Preface

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 underscore 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:

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 v1.6 Essentials Guide (ARM DUI 0181)
- RealView Debugger v1.6 User Guide (ARM DUI 0153)
- RealView Debugger v1.6 Target Configuration Guide (ARM DUI 0182)
- RealView Debugger v1.6 Command Line Reference Guide (ARM DUI 0175)

Refer to the following books in the RVCT document suite for more information on the compilation tools component of RVDS 2.0:

- RealView Compilation Tools Essentials Guide (ARM DUI 0202)
- RealView Compilation Tools Compiler and Libraries Guide (ARM DUI 0205)
- RealView Compilation Tools Linker and Utilities Guide (ARM DUI 0206)
- RealView Compilation Tools Assembler Guide (ARM DUI 0204)
- RealView Compilation Tools Developer Guide (ARM DUI 0203).

The following documentation provides general information on the ARM architecture, processors, associated devices, and software interfaces:

- ARM Reference Peripheral Specification (ARM DDI 0062)
Refer to the following documentation for information relating to the ARM debug interfaces suitable for use with RealView Debugger:

- ARM Agilent Debug Interface User Guide (ARM DUI 0158)
- Multi-ICE™ Version 2.2 User Guide (ARM DUI 0048)
- ARM MultiTrace™ User Guide (ARM DUI 0150).

Refer to the following documentation for information relating to specific ARM Limited processors:

- ARM7DI™ Datasheet (ARM DDI 0027)
- ARM710T™ Datasheet (ARM DDI 0086)
- ARM720T™ Datasheet (ARM DDI 0087)
- ARM740T™ Datasheet (ARM DDI 0008)
- ARM7TDMI™ Technical Reference Manual (ARM DDI 0210)
- ARM7EJ-S™ Technical Reference Manual (ARM DDI 0214)
- ARM9TDMI™ Technical Reference Manual (ARM DDI 0180)
- ARM920T™ Technical Reference Manual (ARM DDI 0151)
- ARM922T™ Technical Reference Manual (ARM DDI 0184)
- ARM9EJ-S™ Technical Reference Manual (ARM DDI 0222)
- ARM926EJ-S™ Technical Reference Manual (ARM DDI 0198)
- ARM940T™ Technical Reference Manual (ARM DDI 0144)
- ARM946E-S™ Technical Reference Manual (ARM DDI 0201)
- ARM966E-S™ Technical Reference Manual (ARM DDI 0213)
- ARM1020E™ Technical Reference Manual (ARM DDI 0177)
- ARM1022E™ Technical Reference Manual (ARM DDI 0237)
- ARM Embedded Trace Macrocell Specification (ARM IHI 0014).

Refer to the following documentation for details on the FLEXlm license management system, supplied by GLOBErotter Inc., that controls the use of ARM applications:

- ARM FLEXlm License Management Guide v3.0 (ARM DUI 0209).

Make sure that you use version 3.0 of this documentation for details on license management in RealView Debugger v1.6.1 for RVDS 2.0.

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:


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 publications 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.
Chapter 1
Introduction to RealView Debugger Extensions

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

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

RealView Debugger is a program that enables you to execute source-level or assembly language programs, and control the flow of program execution. 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 1-3
- *Chapter 5 Working with Multiple Target Connections* on page 1-4.

---

**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 v1.6 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 must obtain the appropriate license before you can use this extension. 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 either:

- setting simple trace control points, ranges, and triggers
- setting 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.
Introduction to RealView Debugger Extensions

Note
It is recommended that you read Examples of using trace in RealView Debugger on page 2-85 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 that is available in RealView Debugger, and describes the benefits and limitations of the level of support it provides. You must obtain the RealView Debugger support package for the RTOS you are using before you can use this extension. Select Goto RealView RTOS Awareness Downloads from the Code window Help menu for information on how to do this.

The chapter shows how to use the thread drop-down list in the Code window and the additional tabs available in the Resource Viewer window. 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 section Using RealView Debugger RTOS support on page 3-4 before attempting to debug an RTOS using RealView Debugger. This section provides two examples of debugging an RTOS, 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. You must obtain the appropriate license before you can use this extension.
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 Trace, DSP, or Multiprocessor extensions, you must have a valid license. The RTOS extension is not licensed by ARM, but you must obtain enabling software for your chosen RTOS. See the Chapter 3 RTOS Support for more information.

You can use all features during the 45-day temporary license period that begins when you first install RealView Debugger, but you must obtain a permanent license for the desired extension before the temporary license period has expired. For details, see the ARM FLEXlm License Management Guide v3.0.

All licensing for RealView Debugger is controlled by the FLEXlm license management system. You can use the FLEXlm server software, running on Windows, Solaris, or Linux platforms, to track and maintain control of your RealView Debugger license (see the ARM FLEXlm License Management Guide v3.0).
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

With the exception of the RTOS extension, there are no additional software requirements to use the RealView Debugger extensions. For information about RTOS software, see Chapter 3 RTOS Support on page 1-3.
1.4 Supported hardware

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

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

1.4.1 Hardware for tracing

The tracing 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 using the RTOS-support 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.
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 2-8
- Configuring the ETM on page 2-12
- Configuring trace capture on page 2-18
- Using the Analysis window on page 2-39
- Examples of using trace in RealView Debugger on page 2-85.
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 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 2-19 for details.

**Complex**

These include AND or OR conditions, conditions on the number of executions, and complex comparisons. See Setting complex tracepoints on page 2-25 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.
Licensing and operating restrictions

RealView Debugger Trace support is a separately licensed component. You must obtain a license from your ARM distributor.

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 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 2-8.

ETM trace solutions

If you are using an 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 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 (ETM9)
- DSP-Group Oak
- DSP-Group TeakLite
- Motorola 56600
- Intel XScale.

**Simulators**

If you do not have trace hardware, you can use one of the following hardware the RealView ARMulator ISS simulator.

### 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
- ARMulator on page 2-6
- OakDSPCore MaxCore Simulator on page 2-7.

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.
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 FIFOFULL, have been connected within the device containing the ETM. To determine this, refer to the documentation that accompanies your device.
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 2-19 for details), but not trace ranges or complex tracepoints.

ARMulator

The resources available for tracing are dependent on the processor variant you choose to simulate:

ETM-enabled ARM processor

The ETM resources available for tracing vary depending on the ETM processor variant you select. See Table 2-1 on page 2-5 for details on the resources available with each ETM configuration type.

To connect to an ETM-enabled processor in ARMulator, you must connect using the ARM-A-RR connection option in the Connection Control window. For details on connecting to targets in this window, see the chapter Connecting to Targets in the RealView Debugger v1.6 Target Configuration Guide.

Note

If you use the ARM-A-RR connection option to connect to ARMulator, and you select a non ETM-enabled processor variant, no trace functionality is available.

Non ETM-enabled ARM processor

In this case, there are no ETM resources to use. Only basic instruction tracing support is available. You can set only trigger points, and start and end tracepoints (see Setting simple tracepoints on page 2-19 for details).

To access these limited trace features, you must connect using the Connection Broker option, rather than the ARM-A-RR connection option, in the Connection Control window. For details on connecting to targets in this window, see the chapter Connecting to Targets in the RealView Debugger v1.6 Target Configuration Guide.

Note

ARMulator does not support non-ARM processors.
OakDSPCore MaxCore Simulator

Only basic instruction tracing support is available. That is, you can set only trigger points, and start and end tracepoints (see Setting simple tracepoints on page 2-19 for details).

To access these limited trace features, you must connect using the Connection Broker option, rather than the ARM-A-RR connection option, in the Connection Control window. For details on connecting to targets in this window, see the chapter Connecting to Targets in the RealView Debugger v1.6 Target Configuration Guide.
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 2-10.

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 2-85. 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. Obtain a license, as described in the ARM FLEXlm License Management Guide v3.0.
2. Ensure that you have the required system components to perform tracing (see System requirements on page 2-3).
3. 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 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 2-9 shows an example of how RealView Debugger can be joined with ARM hardware to enable tracing capabilities.
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
Tracing with RealView Debugger

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 2-85 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 v1.6 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 2-12 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 2-40 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 2-18.

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 2-39 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 v1.6 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, irrespective of the trace capture criteria you set and the number of trace captures you perform (see Configuring trace capture on page 2-18).

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.

---
**Note**
---

This dialog appears only when you are using an ETM-enabled processor variant, either with trace hardware or as simulated by an ARMulator.

The Configure ETM dialog is shown in Figure 2-2.

---
**Figure 2-2 The Configure ETM dialog**
---

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 a buffer size, 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.

This section describes the options available in the Configure ETM dialog, as shown in Figure 2-2 on page 2-12. After you have configured the ETM, click OK to save the configuration, or Cancel to discard any changes.

The options in the Configure ETM dialog are:

**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.

**Trace port mode**

This control specifies the way the trace port is operated. It is set to 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.
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 ASIC. 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.

Data tracing mode

This control specifies the type of information the ETM traces for data transfer instructions. It is set to one of the following values:

- **Trace data only**  Use this option when you want information about the value(s) transferred, but not the data transfer address.
- **Trace address only**  Use this option when you want the data transfer address, but no information about the value(s) transferred.
- **Trace 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 tracing. It enables you to select what information will be captured when tracing is enabled. See Configuring trace options on page 2-40 for information on how to enable tracing.
Trace buffer size

This value represents the maximum number of cycles of trace that is stored by your trace capture hardware. Typically, you set this to the maximum value that your trace capture hardware can support, but you can reduce it to speed up trace decompression. By default, this field shows the currently configured trace capture hardware buffer size.

If cycle-accurate tracing is disabled, the trace capture hardware typically captures between one and two processor cycles for each instruction captured. If cycle-accurate tracing is enabled, it captures every cycle for which tracing is enabled.

—— Note ———

Some logic analyzer solutions might not support a variable buffer size. These always store the maximum number of cycles. Conversely, some logic analyzer solutions might support only certain set buffer sizes. These select the buffer size closest to that requested. Refer to the documentation that accompanies your analyzer for details.

———

Memory map decode

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

———

Processor stalling

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 ASIC can include a facility to stall the processor until the FIFO buffer is empty. To enable this facility, select the option Stall processor to prevent overflow. 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 ASICs. See the documentation that accompanies your ASIC to find out whether FIFOFULL is implemented.

———
Enable Timestamping

Selecting this option enables the timestamp recording logic in the trace capture hardware. Timestamps are displayed in the Time/cycl and +Time columns of the Analysis window (see Column types on page 2-48). You can change the format in which timestamps are displayed using the Scale Time Units... option in the View menu (see Filtering captured information on page 2-66). 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.

However, transmitting the timestamps uses noticeable additional bandwidth on the trace capture hardware to host connection, so only enable this feature when you have to.

Note

If you do not select this option, no time information is returned when using the Profile view in the Analysis window. In this case, the Exec% column contains only the value 0, and no histogram information is displayed.

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

Cycle accurate tracing

This option determines whether the ETM operates in cycle-accurate mode. The available modes are:

Cycle-accurate mode

In this mode, the ETM records the number of cycles executed while tracing is enabled. 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/cycl column, which displays timestamp values, to measure across discontinuities in the trace output.

Note

Timestamp values are displayed only if timestamps are enabled (see the option Enable Timestamping in this section).
Non cycle-accurate mode

In this mode, 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.

Trace Coprocessor

Select this option if you want the data associated with Move Coprocessor from ARM Register (MCR) and Move ARM Register from Coprocessor (MRC) instructions to be traced.

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.

After you have configured the ETM, click **OK** to save the configuration, or **Cancel** to discard any changes.
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 2-19
- Setting complex tracepoints on page 2-25
- Editing tracepoints on page 2-36
- Starting trace capture on page 2-38.

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 2-66).

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 2-25 for details.

4. Select **View → PaneViews → 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 the chapter *Working with Breakpoints* in the *RealView Debugger v1.6 User Guide* for more information on this pane.

5. Execute your program to begin trace capture with the details you have set, as described in *Starting trace capture* on page 2-38. See the section on *Controlling Execution* in the *RealView Debugger v1.6 User Guide* for details.

---

**Note**
The availability of tracepoint options is dependent on the resources you have available. If your FIFO configuration is small, or if you have already used a number of resources, 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 use 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 2-20).

#### 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 2-52). 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 data before, after or about the trigger (see the **Collect data**... options in *Configuring trace options* on page 2-40).

**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 2-52). 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.
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. 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 2-3 on page 2-22. From this dialog, you can select from a variety of options for setting a trigger, tracepoints or trace ranges.
Figure 2-3 List Selection dialog for setting simple tracepoints

Note
The options that appear in this dialog are dependent on the available resources. 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. 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 2-40).

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 2-40).

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 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 2-40).

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
If you set a trace end point but no trace start point, no trace information is returned, and a warning is displayed in the Output pane.

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

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.

——— Note ————

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 \( \triangleright \) 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 \( \triangleright \) 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

Select **Debug → Tracepoints** from the Code window main menu to display the Tracepoints menu, shown in Figure 2-4.

---

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 2-26)
Use one of the four complex tracepoint dialogs to set commonly-used complex tracepoints (see Using the complex tracepoint dialogs on page 2-30).

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 2-5.

![Figure 2-5 Set Address/Data Break/Tracepoint dialog](image)

See the Breakpoints chapter in the RealView Debugger v1.6 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 2-6.

![Figure 2-6 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**

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 2-7 on page 2-28.
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 2-8.
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 2-9.

![List Selection](image)

**Figure 2-9 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 2-20 for detailed information on these options.

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. The following tracepoint 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 \( \rightarrow \) 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 text box contains a value, RealView Debugger takes this value as the start address, and inserts `..` after the value. Enter the end address.

**Address Range by Length**

If you select this option when the 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 text box 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 2-33
- *Trace on X after A==B...* on page 2-34
- *Trace if A==B in X...* on page 2-35.

**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 2-10 on page 2-33.
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 2-11 on page 2-34.
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 2-94 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 2-12.
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 2-13.

![Figure 2-13 Trace if A==B in X dialog](image)

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 2-14.

![Break/Tracepoints pane](image)

**Figure 2-14 The Break/Tracepoints pane**

**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 zoom.

From the Break/Tracepoints pane, you can:
- Modify simple tracepoints on page 2-37
- Disable tracepoints on page 2-37
- Clear tracepoints on page 2-37
- Find corresponding source code on page 2-37.

For general details on using this pane, including how to hide or display it, see the Breakpoints chapter in the *RealView Debugger v1.6 User Guide*. 
Modify simple tracepoints

To modify a simple 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. Edit the tracepoint as required, then click OK to confirm any changes and close the Set Address/Data Break/Tracepoint dialog.

——— Note ————
You cannot use this method to modify a complex tracepoint. If you have set up a complex tracepoint and you want to change it, you must delete it using the Clear option and then set the tracepoint again.

——— Note ————
Disabling a complex tracepoint might have unpredictable results. It is recommended that you delete complex tracepoints using the Clear option instead of disabling them.

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 ✔.

Clear tracepoints

To remove a tracepoint, right-click on the entry 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.

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 being 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 2-40).

   **Note**

   Tracing is enabled by default, but trace data is only be returned you have set a tracepoint or if you have selected Continuous Collection Mode (No Tracepoints) 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 trace options on page 2-40
- Interpreting the data on page 2-47
- Viewing profiling information on page 2-55
- Finding information on page 2-60
- Filtering captured information on page 2-66
- Changing the display on page 2-76
- Saving and loading trace information on page 2-79
- Other window elements on page 2-81.

2.5.1 Overview of the Analysis window

The Analysis window enables you to:

- configure global tracing options
- analyze the results of a trace capture to give profiling information
- view tracing and profiling information, using one of six tabs
- 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 2-15 on page 2-40 shows an example of the Analysis window.
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 both your trace capture hardware and your target processor.

The check marks 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* on page 2-41
- *Select Analysis Configuration...* on page 2-41
- *Connect Analyzer/Analysis...* on page 2-41
- *Configure Analyzer Properties...* on page 2-41
- *Physical to Logical Address Mapping...* on page 2-42
- *Set/Edit Event Triggers* on page 2-42
- *Tracing Enabled* on page 2-43
- *Collect Data Before Trigger* on page 2-43
- *Collect Data Around Trigger* on page 2-43
- *Collect Data After Trigger* on page 2-43
- *Stop Processor on Trigger* on page 2-43
• Stop Processor on Buffer Full on page 2-44
• Stop Collecting on Buffer Full on page 2-44
• Continue Collecting on Buffer Full on page 2-44
• Continuous Collection Mode (No Tracepoints) on page 2-44
• Store Control-flow Changes Only on page 2-44
• Set Amount of Trace Buffer to Read... on page 2-45
• Set Trace Buffer Size... on page 2-45
• Clear Trace Buffer on page 2-46
• Clear All Triggers on page 2-46
• Automatic Update on New Buffer on page 2-46
• Automatic Update on Append to Buffer on page 2-46
• Append New Buffer to Existing on page 2-47.

Copy
This option copies the selected text to the clipboard.

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

Connect Analyzer/Analysis...
If you are using a trace capture hardware device that can be further configured to perform tracing, this option invokes its configuration dialog. This option is enabled only if your target configuration supports this functionality.

If you are connecting using the Connection Broker option, rather than the ARM-A-RR connection option, you must select this option to enable trace.

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 2-16 on page 2-42.
Figure 2-16 DSP Group configuration dialog

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.

**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.

**Set/Edit Event Triggers**

This option enables you to set or edit event triggers.

**Note**

This option is disabled for ETM-based targets.
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.

Collect Data Before Trigger

This option 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 Data 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 Data 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.
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.

Continuous Collection Mode (No Tracepoints)
This option instructs RealView Debugger to collect trace information without requiring you to configure any tracepoints. That is, 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 ---
If you have set a tracepoint, this option does not apply.

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.

**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 2-17.

![Figure 2-17 Setting the amount of trace buffer to read](image)

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.

**Set Trace Buffer Size...**

This option enables you to set the size of the trace buffer. This option is the same as **Trace buffer size** in the Configure ETM dialog. When you select this option, a Prompt dialog is displayed, as shown in Figure 2-18 on page 2-46.
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.

**Clear Trace Buffer**

This option clears the information in the trace buffer.

**Clear All Triggers**

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

**Note**
This option is disabled for ETM-based targets.

**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.

**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.
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.

2.5.3 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:

- **Tabbed view types**
- **Column types** on page 2-48
- **Status lines** on page 2-52.

**Tabbed view types**

Figure 2-19 shows the tabs available at the bottom of the Analysis window.

![View-type tabs](image)

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 2-18.
- **Dsm** Displays the disassembly of traced instructions and data.
- **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.

This 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 2-55.

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 2-76 for information on specifying which columns are displayed.

Available trace view columns are:

Elem
Displays the position of each element within the trace buffer, where the value can represent either:

- An index within the trace buffer.
- A cycle number, if your trace capture device supports cycle-accurate tracing (see the Configure Analyzer... option in Configuring trace options on page 2-40 for details). If you select this option, you must also ensure that the option Cycle accurate tracing is selected in the Configure ETM dialog (see Configuring the ETM on page 2-12).

This column is displayed only when Position is selected in the Trace columns menu.

Time/cycl
Displays the timestamp value. You can change the format of the values by selecting Scale Time Units from the View menu.

This column is displayed only when Absolute Time is selected in the Trace columns menu, and when either Enable Timestamps or Cycle accurate tracing is enabled in the Configure ETM dialog. You can also use the Define Processor Speed for Scaling... option in the Filter menu to scale between times and cycle numbers.
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Time</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 <strong>Scale Time Units</strong> from the <strong>View</strong> menu. This column is displayed only when <strong>Relative Time</strong> is selected in the <strong>Trace columns</strong> menu, and when either <strong>Enable Timestamps</strong> or <strong>Cycle accurate tracing</strong> is enabled in the Configure ETM dialog.</td>
</tr>
<tr>
<td>Type</td>
<td>Displays the access type of the current element, which can be any of:</td>
</tr>
<tr>
<td></td>
<td><strong>Bus</strong>  Bus state change</td>
</tr>
<tr>
<td></td>
<td><strong>Code</strong> Code access (fetch)</td>
</tr>
<tr>
<td></td>
<td><strong>Data</strong> Data access (read or write)</td>
</tr>
<tr>
<td></td>
<td><strong>DMA</strong>  <em>Direct Memory Access</em> (DMA) transfer operation</td>
</tr>
<tr>
<td></td>
<td><strong>Exec</strong> Instruction was executed</td>
</tr>
<tr>
<td></td>
<td><strong>Int</strong>  Interrupt vectoring</td>
</tr>
<tr>
<td></td>
<td><strong>No Exec</strong> Instruction was not executed</td>
</tr>
<tr>
<td></td>
<td><strong>Pin</strong>  Pin state change</td>
</tr>
<tr>
<td></td>
<td><strong>PreF</strong> Prefetch (so not executed)</td>
</tr>
<tr>
<td></td>
<td><strong>Prob</strong> External probe state change.</td>
</tr>
<tr>
<td></td>
<td>If an access type is prefixed by an R, this indicates a read access. W indicates a write access.</td>
</tr>
<tr>
<td></td>
<td>This column is displayed only when <strong>Access Type</strong> is selected in the <strong>Trace columns</strong> menu.</td>
</tr>
<tr>
<td>Address</td>
<td>Displays the address of the instruction or data accessed.</td>
</tr>
<tr>
<td></td>
<td>This column is displayed only when <strong>Address as Value</strong> is selected in the <strong>Trace columns</strong> menu.</td>
</tr>
<tr>
<td>Symbolic</td>
<td>Displays the symbolic information for the current range, and takes the form <code>symbol</code>.</td>
</tr>
<tr>
<td></td>
<td>This column is displayed only when <strong>Address as Symbol/Line</strong> is selected in the <strong>Trace columns</strong> menu.</td>
</tr>
<tr>
<td>Count</td>
<td>For the <strong>Func</strong> 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.</td>
</tr>
</tbody>
</table>
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.

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 symbol+offset. For example, Arr_2_Glob+0x65 might be a data access to the variable address Arr_2_Glob, with an offset of 0x65. Alternatively, this can take the form source_module_name\#line_number. For example, MAIN\#144 might be an access at line 144 that is within the file main(). The 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 2-75).

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 2-58.

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: Debug State**

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

**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: 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.

**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 address tracing until it can resynchronize.

Error: Unable to trace Java state, trace data ignored

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

Error: No data in trace buffer

The trace buffer is composed entirely of zero. This error is very rare. This message only occurs if you are using an XScale target.

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. This message 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.

Warning: 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.
Warning: 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.

Warning: 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.

Warning: 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.

Warning: 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).

Warning: 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.

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: Data address synchronization restored

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

Warning: 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: 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: Memory address unknown, insufficient trace data
This error 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.

2.5.4 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.

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 2-20 on page 2-56.
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 2-58
- **Data interpretation options** on page 2-59.

**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 this 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 2-75). 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 2-58.

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 that is 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 2-21 on page 2-59 shows an example of call-graph information displayed in the Profile tab.
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 2-89 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.

2.5.5 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 2-61
- *Find Time Match...* on page 2-61
- *Find Raw Address Match...* on page 2-62
- *Find Address Expression Match...* on page 2-63
- *Find Data Value Match...* on page 2-64
- *Find Symbol Name Match...* on page 2-65
- *Find Next Match* on page 2-66
- *Find Previous Match* on page 2-66.

**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 2-19.
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 2-48](#)). When you select this option, a Prompt dialog is displayed, as shown in Figure 2-22.

![Figure 2-22 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/cycle** 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 2-23 on page 2-62.
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/Cycl** column. Only an exact match is returned. See *Scale Time Units...* on page 2-75 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/Cycl** column. (See *Scale Time Units...* on page 2-75 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/cycl** 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 2-24 on page 2-63.
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 2-25 on page 2-64.
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 2-26 on page 2-65.
Figure 2-26 Finding a data value match

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

- **Data value** Specify a data value, in decimal or hexadecimal format.

- **Data range** Specify a range of data values, where RealView Debugger displays the first found value that is read from, or written to, memory, within that range. The range you specify can be either:
  - `data_low..data_high`, such as `1..10`, where RealView Debugger locates the first occurrence of any data value within the range 1 to 10 being read from, or written to, memory.
  - `data_low..+len`, such as `40..+10`, where RealView Debugger locates the first occurrence of any data value within the range 40 to 50 being read from, or written to, memory. The `len` of 10 represents the offset value.

**Find Symbol Name Match...**

Enables you to locate a specific symbol-name string, such as a function name or variable, within the **Symbolic** column. When you select this option, a Prompt dialog is displayed, as shown in Figure 2-27.

![Prompt dialog](image)

**Figure 2-27 Finding a symbol name match**

Enter the symbol-name filter, then click **Find**. The symbol name filter can contain the following characters:

- `*` A multi-character wildcard.
- `?` A single-character wildcard.
For example, to find the variable `Ptr_1_Glob`, you might use `Ptr_1_*`. Alternatively, you might use `Ptr_?_Glob`.

RealView Debugger displays the first found symbol name that matches your entry.

**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.

### 2.5.6 Filtering captured information

This section describes the View 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:

- *Update* on page 2-67
- *Clear Filtering* on page 2-67
- *Filter on Position Match...* on page 2-67
- *Filter on Time Match...* on page 2-68
- *Filter on Raw Address Match...* on page 2-69
- *Filter on Address Expression Match...* on page 2-70
- *Filter on Data Value Match...* on page 2-71
- *Filter on Symbol Name Match...* on page 2-71
- *Filter on Access Type Match...* on page 2-72
- *Filter on Percent Time Match...* on page 2-73
- *AND Filters (vs. OR)* on page 2-74
- *Code Window Tracking* on page 2-74
- *Show Details View* on page 2-74
- *Scale Time Units...* on page 2-75
- *Define Processor Speed for Scaling...* on page 2-75
- *Show Position Relative to Trigger* on page 2-76.
Tracing with RealView Debugger

Update

This option refreshes the information in the Analysis window.

Clear Filtering

This option clears any filters that you have set up on the results of a trace capture that has already been performed.

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 2-48). When you select this option, a Prompt dialog is displayed, as shown in Figure 2-28.

![Figure 2-28 Filtering on a position match](image)

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

**Entry position**

Specify an element number, in decimal or hexadecimal format, if you want to filter the information to display only that row.

**Entry position range**

Specify a range of element numbers, where RealView Debugger filters the information down to only rows within that range. The range you specify can be either:

- `low..high`, such as `1..10`, where RealView Debugger displays only those rows containing **Elem** values within the range 1 to 10.
- `low..len`, such as `40..+10`, where RealView Debugger displays only those rows containing **Elem** values within the range 40 to 50. The `len` of 10 represents the offset value. You can also enter a negative value range, such as `-10..10`
Filter on Time Match...

Enables you to filter the information down to a specified timestamp value, or range of
timestamp values, as contained in the Time/cycl 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 2-29.

Figure 2-29 Filtering on a time match

Enter one of the following types of information, then click Filter:

**Time value** Specify a timestamp value, in the format currently used for the
Time/Cycl column, if you want to filter the information to display only
that row. See Scale Time Units... on page 2-75 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 2-75 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/cycl 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 2-30.

![Figure 2-30 Filtering on a raw address match](image)

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.

---

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 2-31. An address expression can be a function, structure, or array symbol.

Figure 2-31 Filtering on an address expression match

Enter one of the following types of information, then click Filter:

Address expression
Specify an address expression if you want to filter the information to display only that row. An address expression can be one of:
- a function name
- a structure name
- an array symbol.

Auto-range
An auto-range of address values is generated from an expression that you enter, which can be any of:
- A function name, where the generated address range is from the start-to-end of the function.
- A structure, where the generated address range is from the start-to-end of the structure.
- An array symbol, where the generated address range is from the start of the variable to the end, where the end is the start+sizeof(var). For example, if the start value of where the array is stored in memory is 0x8000, and the array size is 16 bytes, the end address is considered to be 0x8010 (that is, 0x8000+16).

RealView Debugger filters the information down to only rows represented by the generated auto-range.
Filter on Data Value Match...

Enables you to filter the information down to a specified data value, or range of data values, that is read from, or written to, memory. These values can be found in either of the Data columns (decimal or hexadecimal). When you select this option, a Prompt dialog is displayed, as shown in Figure 2-32.

![Figure 2-32 Filtering on a data value match](image)

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 2-33 on page 2-72.
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 2-34.

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
Tracing with RealView Debugger

- 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 these options 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 2-35.

![Figure 2-35 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 \texttt{low..high} range of percent values, such as \texttt{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 \texttt{low..+len} range, such as \texttt{40..+10}, where RealView Debugger
displays only those rows containing percentage-of-execution values
within the range 40\% to 50\%. The \texttt{len} of 10 represents the offset value.

\textbf{Note}

You can also use floating point values such as 40.5.

\textbf{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 \textbf{Filter on Position Match...} and a \textbf{Filter on
Data Value Match...}, the following would apply, depending on whether the \textbf{AND
 Filters (vs. Or)} option is selected:

\begin{itemize}
  \item 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
  \item 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.
\end{itemize}

\textbf{Code Window Tracking}

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

\textbf{Show Details View}

Displays the detailed information in the details view at the bottom of the window (see
\textit{Other window elements} on page 2-81). To hide the status-bar details, deselect this
option.
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.

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 **Time/cycle** column. When you select this option, a Prompt dialog is displayed, as shown in Figure 2-36.

![Figure 2-36 Defining processor speed](image)

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

2.5.7 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 the Trace columns menu enable you to indicate the columns you want to display or hide. To display a particular column, select it (✓) in the Trace columns 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 2-48.

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/cycl** 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/cycl and +Time columns in hexadecimal format.

Data Value in Decimal
Displays all values in the Time/cycl 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.

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.

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/cycl 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.

2.5.8 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... on page 2-80
- Save Filtered Trace Buffer to File... on page 2-81
- Close Loaded File on page 2-81
- Print Trace Lines... on page 2-81
- Exit Window on page 2-81.

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 following 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 View menu, this option ensures that you store the post-filtered trace information.

Close Loaded File

This option closes the file that is currently loaded in the Analysis window, and clears the trace information from the window.

Print Trace Lines...

This option enables you to print the trace-buffer contents contained in the Analysis window. When you select this option, a standard Print dialog is displayed.

Exit Window

This option closes the Analysis window.

2.5.9 Other window elements

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

- Details view
- Context menu on page 2-83.

Details view

The details view is located directly above the tabs at the bottom of the Analysis window, as shown in Figure 2-37. To display status-bar information, select the option Show Details View from the View menu.

Figure 2-37 Details view
The details view provides different information depending on the tab currently displayed:

**Square-bracket information**

Contains up to three values representing the following:

- The number of elements shown
- The number of elements collected by RealView Debugger
- The number of elements in the analyzer buffer.

When you filter the trace information, the first number is used to show how many elements you have selected by filtering. When you collect only the first \(N\) elements of the hardware trace buffer, the second number shows the amount of elements that have been collected. RealView Debugger numbers elements from \(-M\) to \(+N\), where 0 is the trigger point.

--- Note ---

In cases where there is only a single value shown in the brackets, RealView Debugger has read all the trace data, and the number in the brackets is the number of elements shown.

In cases where there are only two values shown in the brackets, RealView Debugger has all the trace data that is available, but the number of elements shown is different from the number collected. This is represented as \([\text{number}_\text{shown}, \text{number}_\text{collected}]\).

---

**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-click)

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. 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 automatic tracking, deselect the option Track in Code Window (double right-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 2-38).

![Figure 2-38 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 2-60 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 2-60 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
- Capturing profiling information on page 2-89
- Setting up a complex tracepoint on page 2-94.

2.6.1 Introduction to the examples

It is recommended 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 \Examples directory of your RealView Debugger installation:

- dataabort.axf
- dhrystone.axf.

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

Before beginning each example, start RealView Debugger and ensure that your trace hardware or simulator and target are properly configured.

This following examples are provided:

- Finding the cause of a data abort
- Capturing profiling information on page 2-89
- Setting up a complex tracepoint on page 2-94.

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.

This example requires you to have the following RealView Debugger-supported trace hardware components, which must be installed and connected properly:

- a JTAG interface unit
- trace capture hardware
- 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.

You cannot perform this example using a simulator, because simulators do not cause a data abort.

The DataAbort 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 dataabort.axf into the debugger. This file is located in the \Examples directory in your root installation. The tab for dataabort.cpp is displayed in the File Editor pane.

2. Enable data aborts in the vector catch as follows:
   a. In the Register pane of the Code window, click the Debug tab. Debugger internal values are displayed.
   b. Double-click on the vector_catch value, and change the value to 0. 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 Collect Data Before Trigger from the Edit menu of the Analysis window.
   c. Select 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.
Tracing with RealView Debugger

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 2-39.

Figure 2-39 Setting a trigger point
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 2-40. 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 2-41.

![Figure 2-41 Results displayed in the Dsm tab](image)

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 needed, 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.
This example requires you to have the following RealView Debugger-supported trace hardware components, which must be installed and connected properly:
- a JTAG interface unit
- trace capture hardware
- 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.

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 \Examples directory in your root installation. 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 \( \mathbb{3} \) 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 2-42 on page 2-91.
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 2-19.

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 2-43.

![Figure 2-43 Results displayed in the Profile tab](image)

In the **Profile** tab, as shown in Figure 2-43, 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 2-48.

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 2-44 on page 2-93.
Figure 2-44 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 2-45 shows the data for a single function, Func_2.

Figure 2-45 Call-graph data for 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 2-89. In this example, assume that you want to trigger trace in Proc_2, but only after 50 executions of the function Proc_1. For this example, you should use ARMulator. You can perform the example using trace hardware, but in some cases the results might be inaccurate.

This example requires you to have ARMulator installed (see Appendix B *Setting up the Trace Software* for details on connecting to ARMulator).

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 2-46.

![Figure 2-46 Trace on X after Y executed N times dialog](image)

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.

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 2-47.

![Figure 2-47 Break/Tracepoints pane with complex tracepoints set](image)

5. View the Analysis window by selecting View → Analysis Window from the Code window. Ensure that Collect Data Before Trigger is checked 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. In the Analysis window, select the Profile tab, as shown in Figure 2-48.

You can see from the Count column that Proc_2 has been executed 50 times, as specified.
Chapter 3
RTOS Support

This chapter describes the Real Time Operating System (RTOS) support that is available in RealView Debugger. It contains the following sections:

- About Real Time Operating Systems (RTOSs) on page 3-2
- Using RealView Debugger RTOS support on page 3-4
- Associating threads with views on page 3-8
- Using the Resource Viewer window on page 3-12
- Working with threads in the Process Control pane on page 3-14
- RTOS resource CLI commands on page 3-15.
3.1 About Real Time Operating Systems (RTOSs)

Operating systems provide software support for application programs running on a target. RTOSs are operating systems that are designed for computers that interact with real-world activities, and so the treatment of time is critical to successful operation. For example, if an application is in control of a car engine 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:

- a process that models the motion of the cylinder, enabling it to control ignition and valve timing
- a process that monitors fuel consumption and car speed and displays trip distance and fuel economy on the dashboard.

Using components like this enables the RTOS to ensure that the job that must be done next to meet the deadlines can be scheduled next. RTOS processes can either:

- Share only specifically designated memory, normally dedicated to inter-process communication. These are called *processes.*
Implicitly share the memory of the processor, so that each can share all the data and code of the others. These are called *threads*.

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 this as different processes, some of which have many threads.

### 3.1.1 Debugging an RTOS application

Debugging real-time systems presents a range of problems. This is especially true where the particular software component being debugged interacts with physical hardware, because you normally cannot stop the hardware at the same time as the software.

To avoid having to halt the software, real-time debuggers often support access to the application using software that can be scheduled along with other tasks in the system. This enables the RTOS to maintain the system deadlines. RealView Debugger support for this type of debugging is dependent on the specific RTOS support for your system, and so might not permit you to stop threads independently.
3.2 Using RealView Debugger RTOS support

This section describes the process of configuring RealView Debugger so that it can interpret an application image file correctly on an RTOS target.

--- Note ---

RTOS specific support files are not installed with the product. Instead, you must download and install the support you require. Click Help → Goto RTOS Awareness Downloads from the RealView Code window for more information.

This section contains the following sections:

- Enabling RTOS support
- Basic RTOS configuration options
- Connecting to the target and loading an image on page 3-6.

3.2.1 Enabling RTOS support

Your RTOS support library documentation contains the instructions for installing and configuring the software. The RTOS support library might also include a Board/Chip definition file, *.bcd, containing the information required to enable RTOS support in the debugger.

If you have an RTOS-specific *.bcd file, you can enable RTOS support on your target by referencing the *.bcd file from your current target configuration file, for example rvdebug.brd. Do this using the BoardChip_name entry as described in the configuring custom targets chapter of the RealView Debugger v1.6 Target Configuration Guide.

If you do not have an RTOS-specific *.bcd file, configure the RTOS on your target using your RTOS support library documentation coupled with the information in Basic RTOS configuration options.

3.2.2 Basic RTOS configuration options

RTOS operation is controlled by the RTOS settings group in the Advanced Information block in your current board file. Select File → Connection → Connection Properties... to display the Connection Properties window where you can configure these settings, shown in Figure 3-1 on page 3-5.

--- Note ---

You do not have to configure these options if your RTOS vendor has supplied an RTOS specific *.bcd file.
There are descriptions of the general layout and controls of the RealView Debugger settings windows, which include the Connection Properties window, in the RealView Debugger online help topic Changing Settings.

Note

Do not configure the board file with the debugger connected to a target.

The RTOS group contains the settings:

- **Vendor** Selects the type of RTOS that the application is using. This is an enumerated value that selects the RTOS you are using from the installed support libraries.

- **Load_when** Selects when the debugger loads the RTOS support library for your RTOS. The RTOS features of the debugger are not enabled until the RTOS support library has found the RTOS on your target, and this might mean that you must run the image startup code. When the DLL is loaded, it can:
  - check immediately for the RTOS, with Load_when set to connect
  - wait until an image is loaded before checking for the RTOS, with Load_when set to image_load.

Refer to the documentation for the RTOS support library for more details.
RTOS Support

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

3.2.3 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 these sections:

- Before connecting
- Connecting interactively
- Connecting from the system command line on page 3-7.

Before connecting
Ensure that:

- You compile your RTOS application with debug symbols enabled, so that the debugger can find the data structures it requires.
- You have configured your connection to be RTOS aware as described in either:
  - Enabling RTOS support on page 3-4
  - Basic RTOS configuration options on page 3-4.

Connecting interactively
To connect using the GUI:

1. Start RealView Debugger.
2. Connect to the target using the RTOS-enabled connection, for example using the Connection Control window.
3. Select File → Load Image... to load the image.
   Uncheck the Replace Existing File(s) option in the Load File to Target dialog, or use the ADDFILE and RELOAD commands, if you have several executable files to load.
Connecting from the system command line

You can connect to the target and load your image when the debugger starts up. To do this:

1. Ensure that your workspace specifies an open connection, so that the debugger automatically connects on startup.
   
   If your workspace does not specify an open connection, when you try to load an image a dialog is displayed asking if you want to make a connection, for example using the Connection Control window. If you do, as soon as the connection is made 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 support library for more information. The RealView Debugger command-line syntax is as follows:

```
rvdebug.exe -exec image.axf;target_name;[arg1 arg2 arg3 ...]
```

where:

- `image.axf` Specifies the image loaded when RealView Debugger starts up.
- `target_name` Specifies the target loader information. Where supported, this is the target name that is passed to RTOS image loader. Can be empty.
- `args` Specifies an optional, space-separated, list of arguments to the image. Where several arguments are provided, the list must be enclosed in quotes. The case of arguments is preserved.

See RealView Debugger v1.6 User Guide for more information on RealView Debugger operations:

- getting started and connecting to a target from the command line
- working with images from the command line
- configuring workspaces.

For details of the CONNECT, LOAD, ADDFILE, and RELOAD commands see RealView Debugger v1.6 Command Line Reference Guide.
### 3.3 Associating threads with views

With the image running on your RTOS target, you can start to work with threads in the RealView Debugger Code window. To do this you use the **thread list**. This is described in the following sections:

- Selecting threads from the thread list
- The current thread
- Attaching and unattaching windows
- Manipulating registers and variables

#### 3.3.1 Selecting threads from the thread list

With the RTOS extension loaded into RealView Debugger, the **Thread** button is enabled on the Actions toolbar in the Code window. Click on this button to cycle all unattached windows through the threads list. This sets the current context to the selected thread and changes the code view. The **Thread** button enables you to view each of the active threads in turn without affecting window attachment.

**Note**

If the current Code window is an attached window, a message dialog *Do you want to detach this window* is displayed. Clicking **Yes** causes the window to be detached and a new thread to be displayed. Clicking **No** aborts the action. See **Attaching and unattaching windows** for details on windows attachment.

Clicking on the drop-down arrow causes RealView Debugger to fetch the list of active threads from the target and display a summary, as in the example in Figure 3-2.

![Figure 3-2 Example thread list](image)

Below the menu spacer line are the details of the threads running on the target. For this example, the fields that are displayed for the threads shown in Figure 3-2 are:

- the address of the thread control block
- the name of the thread
- the current priority of the thread
3.3.2 The current thread

An asterisk * beside the name of the thread in the thread list indicates the current thread. The current thread is initially set to the thread that was running on the processor when it stopped. Unattached windows always display the current thread.

You can set the current thread by clicking the Thread button on the Actions toolbar, to cycle through the threads until the desired thread is selected.

When the current thread is changed, for example when you stop the target with a different thread active, a message is displayed in the Cmd tab of the Output pane displaying details of the new current thread. The details include the thread number in decimal and the thread name, if it is available.

See Working with threads in the Process Control pane on page 3-14 for other ways of working with the current thread.

Note

Selecting a thread from the Thread button drop-down list does not set that thread to be the current thread, although it does display that thread.

3.3.3 Attaching and unattaching windows

If you are licensed to use multiprocessor debugging mode, Code windows in RealView Debugger have two states relating to threads and processes. The attachment status is displayed in the title bar after the name of the target:

[Unattached]

All unattached windows display details of the current thread, that is the thread that was most recently running on the target when the target stops.

<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.
Use the thread list, from the Thread button drop-down arrow, to select threads. By default, new Code windows are unattached, and the first menu item entry in the thread list is Attach Window to a Thread. The menu item is ticked when you display this list from an attached window. Select it again to unattach an attached window.

--- Note ---

To display a new Code window, select View → New Code Window.

See Working with threads in the Process Control pane on page 3-14 for other ways of changing window attachment.

### 3.3.4 Manipulating registers and variables

With a thread selected into a Code window, the Register pane displays the registers associated with the thread, which might not have the same values as the current processor registers.

You can change the value in a register in the usual way (as described in the RealView Debugger v1.6 User Guide), but the changed value is not necessarily written into the processor register. Instead, unless the thread is currently running on the processor, it is written to the RTOS Task Control Block (TCB) for the selected thread. When that thread is next scheduled by the RTOS scheduler, the registers used by the new thread are read from the TCB into the processor.

When a thread is interrupted, the Code window displays the code around the last Program Counter (PC) addresses of the thread. This is not necessarily the last PC address of the processor. Similarly, the Stack Pointer (SP) is that of the thread. You can therefore display and change the local variables of the thread.

If you are debugging ARM code, the ARM-Thumb Procedure Call Standard (ATPCS) 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, and so 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:

- 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
- declare the variable to be volatile and recompile the program.

---
3.4 Using the Resource Viewer window

The thread information displayed in the thread list might be supplemented by information in the Resource Viewer window. Select View → Resource Viewer Window to display this window for your current image, shown in Figure 3-3.

The Resource Viewer window title bar reflects the title bar of the calling Code window. In this example, the Code window is attached to the current thread.

The Resource Viewer window includes the:

- File menu
- Help menu
- Resources list
- Details area.

For details on these menus see the chapter describing the desktop in RealView Debugger v1.6 Essentials Guide.

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, an RTOS-specific tab is added to display the processes or threads that are configured, shown in Figure 3-3. Other tabs might be included to support the display of other RTOS data structures, also shown in Figure 3-3.
With the RTOS-specific thread tab visible, the Resource Viewer window shows the same threads list available from the Thread button drop-down list, with the same fields shown. The information displayed is RTOS specific.

If you double-click on one of the lines in the Resources list, to select one of the threads, the lower pane, the Details area, displays more information about that thread.

For more information about the meaning of the fields displayed in the Resource Viewer window, refer to the user manual for your RTOS.

--- **Note** ---

Different RTOS support libraries might display information in different ways in this window, and other RealView Debugger extensions might add other tabs to the Resource list.

---

---
3.5 Working with threads in the Process Control pane

The **Thread** tab in the Process Control pane also displays information about the threads that are running on the target. The information that is displayed is RTOS support library specific. Each configured thread is listed with some summary information available by expanding the tree, shown in Figure 3-4.

![Figure 3-4 The Thread tab in the Process Control pane](image)

In this example, the Process Control pane is floating. This means that the title bar reflects the title bar of the calling Code window.

Each thread is accompanied by an icon that indicates the attachment status of the thread:

- ![This window is attached to this thread.](icon)
- ![The thread is attached to another window, or it is unattached.](icon)
- 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:

**Make Current** Make this thread the current thread (see *The current thread* on page 3-9).

**Attach Window to** Attach the Code window to this thread (see * Attaching and unattaching windows* on page 3-9).
3.6 RTOS resource CLI commands

There might be one or more thread-specific CLI commands defined by the RTOS support library you are using that supplement the GUI thread tools.

These commands use a standard command name style:

Dresource_LIST=expression

where:

resource The name of an RTOS resource, in the same way that the Resource Viewer tabs are named.

expression Is a valid expression identifying the specific instance of the resource.

Refer to your RTOS support library documentation for more details.
Chapter 4
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 4-2
- Using the DSP on page 4-3.
4.1 About DSPs and RealView Debugger DSP support

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. The support includes:

- Oak and TeakLite instruction disassembler
- COFF image file format support for the DSP Group Oak and TeakLite toolchain
- register definitions for Oak and TeakLite
- support for JTAG debug of Oak and TeakLite processors.

You make a connection to a DSP Group processor 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 4-3 for more details.

If you are using a DSP in conjunction with another processor, refer to Chapter 5 Working with Multiple Target Connections for information about using RealView Debugger with multiprocessor systems.

For more information on managing connections, see the chapter describing connecting to targets in RealView Debugger v1.6 Target Configuration Guide.

4.1.1 Licensing and operating restrictions

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

The DSP support in RealView Debugger is invoked by connecting the debugger to an suitable processor. This can be a simulated target, for example using the OakDSPCore MaxCore Simulator, or target hardware.

The DSP extension license includes the OakDSPCore MaxCore Simulator. This enables you to run and debug DSP programs without access to target hardware.

4.2.1 Connecting to the simulator

To access the DSP 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 new_OAK entry to start a simulator connection.
   
   The connection list expands to show your new connection, for example SimOAK_1. The Code window title bar is updated to show this connection and prompts you to load an image.
5. Click **File** → **Load Image**... to load an executable file suitable for the DSP you are using.

If you do not see a new_OAK entry in the Connection Control window, you have not included DSP support in the RealView Debugger installation options. Choose a Custom installation to install DSP support if you want to use the OAK or TeakLite DSP, or the Motorola M56621 DSP.

Failing to connect

If you do see a new_OAK entry 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
```

4.2.2 Connecting to target hardware

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 the chapter describing configuring custom connections in RealView Debugger v1.6 Target Configuration Guide.
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 systems run on more than one target.

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 5-10
- Display coherency on page 5-30
- Processor execution synchronization on page 5-38.
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. The debugger is shipped with a temporary license that enables you to try out these facilities before you buy a permanent license. After this expires you must obtain a permanent license to continue using 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 5-10 describes the RealView Debugger interface.

Display coherency on page 5-30 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 5-38 describes how you 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 using either:
- a single target hardware connection (for example, a JTAG scan chain)
- 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. Figure 5-1 shows the relationship between a single processor (in this instance, an ARM processor core), the debugger and a single Code window.

![Figure 5-1 The relationship of one Code window to a processor](image-url)
The diagram shows the relationship of these components:

**Code window**
Part of the GUI, this component 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 actions 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**
Describes how the debugger connects to the debug target, any information required to use that connection, and what kind of processor the target is using. It might also include cached copies of processor registers or memory.

**Target debug interface**
Uses appropriate workstation hardware, for example a parallel port, and JTAG interface hardware such as ARM Multi-ICE, to enable the debugger to control the processor.

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

The configuration shown in Figure 5-1 on page 5-3 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.

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 operations, for example read register, to retrieve the information required by the display you have selected, and to change the target state, for example, write to a variable, in response to your requests.

  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 window attachment options, described in *Using the Connection menu* on page 5-16. 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.

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.

Figure 5-1 on page 5-3 shows your first connection to the ARM processor on the debug target board. Now you 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.
If you compare Figure 5-1 on page 5-3 with Figure 5-2 on page 5-5, 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 previous ARM connection is still available.

The term current connection is used to denote the connection that the debugger displays unless you override it. In Figure 5-1 on page 5-3, the ARM connection is the current connection. If you then connect to a second connection, the DSP, this becomes the current connection, shown in Figure 5-2 on page 5-5. A RealView Debugger Code window always displays information relating to the current connection unless you attach the window to another connection. See Using the Connection menu on page 5-16 for details on attaching windows.

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, as shown in Figure 5-3.

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.

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 clicking on the Connection button to cycle through the connections in turn.

You can also make a connection to a processor running an RTOS on your debug target board. Figure 5-2 on page 5-5 also shows a CPU running an RTOS supporting four threads, represented by four small circles. This capability is described in more detail in Chapter 3 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.

Following on from Figure 5-2 on page 5-5, Figure 5-4 shows the state of the debugger after you make four connections and create a second Code window.

Figure 5-4 shows:

- a connection to one of the other threads on the RTOS and the connection pointer has moved on to display this in the Code window.
• a new Code window and a new connection displaying the state of the DSP processor on the target.

For more information on selecting connections and the difference between attached and unattached windows, see Using the Connection menu on page 5-16.

5.2.1 Using target debug interfaces

In these descriptions, the target debug interface unit has been ignored. However, the way in which a connection is made is important, not least in the limitations it can impose on you. For example, each debug target interface is capable of supporting different numbers of connections, and different kinds of connections. Similarly, the type of processor that you can connect to varies depending on the underlying debug target interface.

Figure 5-5 on page 5-9 shows a configuration of RealView Debugger that is nearly identical to that in Figure 5-4 on page 5-7. The difference is that there are now three distinct debug target interfaces rather than just one.

Not only do the architectures look similar but the behavior of RealView Debugger does not change, whether there is a single target connection or many connections. However, limitations inherent in these interfaces might mean, for example, that the speed of download changes or that some features might not be available on some interfaces.

Sometimes you might use multiple target debug interfaces to a single target. For example, in the model shown in Figure 5-5 on page 5-9:

• The Multi-ICE interface unit supports the ARM processor.

• If the CPU running the RTOS is an Intel StrongARM, you cannot use a JTAG debug interface to connect to it. Suitable target debug interfaces include the ARM Angel debug monitor.

• The DSP must be driven by a target debug interface that supports DSP debugging, for example the Multi-ICE direct connect OCD-based emulator.
Figure 5-5 Multiple connections, views and debug interfaces to a target
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 includes:
- Using Multi-ICE with multiple targets
- The Connection Control window
- Viewing connection details on page 5-13
- Using the Connection menu on page 5-16
- Disconnecting from targets on page 5-22
- Working with projects on page 5-25.

5.3.1 Using Multi-ICE with multiple targets

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 with the debugger connected to a target.

5.3.2 The Connection Control window

Like 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 5-6.

![Connection Control window in multiprocessor mode](image)

Figure 5-6 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.

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

Expand the entries in the **Name** column to see the available connections, shown in Figure 5-6.

**Establishing connections**

In single processor debugging mode, you make your first connection to a debug target as described in *RealView Debugger v1.6 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 the connection to the list of active connections. Your new connection automatically becomes the current connection and the Code window title bar is updated to show the connection details, shown in Figure 5-7.

![Figure 5-7 Code window title bar showing current connection](image)

See Viewing connection details on page 5-13 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 v1.6 Target Configuration Guide for more details on using this window.

**Current connection**

You can view all your connections from the Code window using the Connection button on the Actions toolbar. Click on the arrow on the button to display the drop-down menu, shown in Figure 5-8. Select the particular connection you want to view.

![Figure 5-8 Connection menu showing connections](image)

See Using the Connection menu on page 5-16 for details on using this menu.

**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 5-9 on page 5-13.
5.3.3 Viewing connection details

As you establish new connections, view details in:

- The Code window title bar
- The Resource Viewer window on page 5-14.

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 5-10.

![Figure 5-10 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.

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

In RealView Debugger, a project can be associated with a connection, that is it is bound to that connection, shown in Figure 5-10.

See Working with projects on page 5-25 for details on project binding.
The connection, including the target processor, the connection number, and the execution vehicle.

[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 5-16 for details.

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 5-11 on page 5-15.

   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 the connection.

4. Select **File → Display Details** from the menu bar to see connection details, shown in Figure 5-11 on page 5-15.
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 from the Actions toolbar, 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.

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.
5.3.4 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 5-12.

![Figure 5-12 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 5-12. The Code window title bar reflects the unattached state of the window, for example:

RVDEBUG(dhry) = @ARM940T_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(dhry) = @ARM940T_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-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 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.
Window attachment

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

**Attached** Only displays information about the attached connection.

**Unattached** 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. This window might also be attached to a thread.

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 **Threads** menu to change this.

See [Attaching windows to multiple connections](#) on page 5-19 for details on configuring window attachment.

Changing the current connection

The list of active connections is extended as you make connections to your target. 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, as shown in Figure 5-12 on page 5-16. 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 5-13.

You can create new Code windows from the default Code window or from each new Code window as it is created. 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 targets and then to attach Code windows to each of these connections. In this example, the same image has been loaded to each connection but this is not necessary to manipulate window attachment.

To attach windows to multiple connections:

1. Start RealView Debugger to display the default Code window (unattached).
2. Connect to three different target processors and load your images, as described in The Connection Control window on page 5-10.
   As each new target becomes the current connection, the title bar in the Code window changes to reflect the new connection and the Color Box changes color. For example, making the last connection current changes the title bar to:
   RVDEBUG(dhry) = @ARM966E-S_1:ARM-A-RR [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 attach it to the current connection by clicking on the Connection drop-down arrow and selecting Attach Window to a Connection.
   The title bar is updated to show that this window is now attached to the current connection, the last connection made, shown in Figure 5-14. The Color Box also changes color.

   Figure 5-14 Code window showing current connection and attachment

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 all unattached windows, that is the second Code window, RVDEBUG_1, shown in Figure 5-15, and the third Code window, RVDEBUG_2.

3. Attach the second Code window, RVDEBUG_1, to the current connection by clicking on the Connection drop-down arrow and selecting Attach Window to a Connection.

This action updates the title bar of the second Code window to show the attachment, shown in Figure 5-16.

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, shown in Figure 5-17.

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 by clicking on the Connection drop-down arrow and selecting Attach Window to a Connection. This action updates the title bar of the third Code window to show the attachment, shown in Figure 5-18.

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 as shown in Figure 5-19.

![Figure 5-19 Resource Viewer showing multiple connections](resource-viewer.png)

The Resource Viewer window has inherited its attachment from the calling window and so the title bar and the Color Box reflect the parent window.

The Conn tab also shows that ARM920T_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. Displaying 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.

See Working with projects on page 5-25 for details on attaching windows when working with projects.

Changing window 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 window attachment in this way does not change the current connection.
5.3.5 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.

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 5-23
- Setting disconnect mode on page 5-24.

Disconnecting from the File menu

When working with multiple targets, you can disconnect from the current connection by selecting File → Connection → Disconnect from the Code window main menu. 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 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 RealView Debugger v1.6 Extensions User 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 chosen connection for termination is the only remaining connection, then this is the same as disconnecting in single processor mode. See RealView Debugger v1.6 Target Configuration Guide for details. In multiprocessor mode, however, the Connection button is disabled on terminating the last connection.

Setting disconnect mode

You can control the way a processor is left when a connection is terminated. This is useful when debugging multiprocessor systems or multithreaded applications.

See the chapter describing connecting to targets in RealView Debugger v1.6 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_A_RR vehicle, to see the Vehicle context menu, shown in Figure 5-20.

Figure 5-20 Vehicle context menu
In multiprocessor mode this 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.

See *Working with projects* for details on managing your connections when working with projects.

### 5.3.6 Working with projects

If you are licensed to work in multiprocessor debugging mode, you can work with:

- multiple connections
- multiple Code windows
- multiple projects
- different window attachment.

This section describes:

- *Working with multiple connections*  
- *Working with attached windows* on page 5-27
- *Project binding with multiple connections* on page 5-27
- *Connecting and disconnecting* on page 5-29.

______ **Note** ______

This section assumes that you are familiar with the concepts and terms explained in the chapter describing managing projects in *RealView Debugger v1.6 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 default active project is shown in the default Code window title bar. If you are not connected, this is the last project that you open. If you are connected, this is the last project that binds successfully.

• The default 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. This is the active project for the 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 5-21.

![](image)

**Figure 5-21 Working with the Project Control dialog box**

Use this dialog box to unbind or rebind projects in the usual way. See the chapter describing managing projects in *RealView Debugger v1.6 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 multiple target connections

When you are working with multiple connections, project operations are relative to the current connection and depend on window attachment. This means that

- By default, the active project is shown at the top of the open project list.
- 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 current 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 managing projects in *RealView Debugger v1.6 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 and an Oak DSP core. The Oak DSP is now 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 [Board]

6. Attach the second Code window, RVDEBUG_1, to the connection to the Oak DSP core. The title bar shows, for example:
Working with Multiple Target Connections

RVDEBUG_1 = @SimOAK_2:Sim [Board]

7. Leave the third Code window, RVDEBUG_2, unattached. The title bar shows the current connection, for example:
   RVDEBUG_2 = @SimOAK_2:Sim [Unattached]

8. Select Project → Open Project... from the default Code window main menu.

9. Locate the required project, for example dhrystone.prj into the debugger. This file is located in the \Examples directory in your root installation.

10. 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 this connection:

   RVDEBUG(dhrystone) = @ARM940T_0:ARM-A-RR [Board]
   RVDEBUG_1<dhrystone> = @SimOAK_2:Sim [Board]
   RVDEBUG_2<dhrystone> = @SimOAK_2:Sim [Unattached]

   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 only the ARM connection. Because the project currently bound to that connection 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 connection 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. This means that 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. This means that you can continue to make changes to the project properties. This applies to user-defined projects and auto-projects.
5.4 Display coherency

This section describes how RealView Debugger is affected by multiprocessor and multithreaded debugging. It is split into the following sections:

- Resource sharing and debugger consistency
- Saving and restoring your .brd file on page 5-31
- Defining shared memory regions on page 5-31.

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 access 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 suffer from resource sharing issues. However, when resources (such as memory) are shared between the processors of different connections to a multiprocessor system, the debugger must address the problem of ensuring the 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:

- To ensure that the change affects all connections accessing the shared memory.
- To 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.

To resolve this, RealView Debugger ensures that changes to a given connection cause an update for each related connection provided that the shared memory area is described to the debugger. 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, for example in `\home\user_name\`, where `user_name` is your Windows login or user name. 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 need 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 v1.6 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 5-22.

![Figure 5-22 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 illustrate how you use them:

- *Defining memory for a symmetric multiprocessor* on page 5-33
- *Defining memory for a three processor multimedia system* on page 5-33.
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 (as shown in the example in Figure 5-22 on page 5-32) like this:

1. Set the value of Start = 0.
2. Set the value of Length = 0xFFFFF.
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 5-23 on page 5-34, 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 as shown in Figure 5-23 on page 5-34. 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 5-22 on page 5-32
- creating a Board/Chip definition file, similar to the AP.bcd shown in Figure 5-24 on page 5-34, and then reference it from the board file using the BoardChip_name setting in the CONNECTION= group.
Using the BoardChip_name method makes the configuration more flexible, but it is slightly more complex to set up.

To configure RealView Debugger for the target shown in Figure 5-23, you must set up several memory blocks. See RealView Debugger v1.6 Target Configuration Guide for more information about setting up target configurations.
Each processor has a memory blocks for its private area and a block for a shared communication area. The ARM920T 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:
  
  - 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 named memory regions, in the example 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, 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 using a direct access mechanism such as sharing the memory address bus.
The memory region is shared using an indirect access method, for example, using a distributed memory controller with a high speed serial bus connecting the processors.

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.
5.5 Processor execution synchronization

When you have multiple processors that are cooperating on 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. This section contains the following sections:

- About execution synchronization
- Synchronization facilities on page 5-43.

5.5.1 About execution synchronization

When several processors are working on the same application, you might decide that you need 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 contains the following sections:

- Terms
- Synchronization and cross-triggering on page 5-42.

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 5-25 shows three processors stopping in response to an external event, such as clicking a stop button.

![Figure 5-25 User halt stopping skid](image)

The skid shown here means that when the debugger inspects the program running on CPU1 it will be more advanced than that on CPU0. The delays involved in this sequence are explained in Table 5-1.

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 small 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.

The processor that initiated the stop, for example because it triggered a breakpoint, stops almost immediately, but others can take longer. If there is cross-triggering hardware on the target, a sequence similar to Figure 5-26 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.

![Figure 5-26 Breakpoint stopping skid using hardware synchronization](image)
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 5-27.

![Figure 5-27 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>
</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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₄</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>t₅</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>t₆</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>
It is normal that a multi-threaded 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.

---

### 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* on page 5-44
- *Execution controls* on page 5-44
- *Cross-triggering controls* on page 5-44
- *Synchronizing processors* on page 5-45
- *Working with synchronized processors* on page 5-46.

**The Synch tab**

Select **File → Connection → Synchronization Control...** to display the connection window with the **Synch** tab displayed, shown in Figure 5-28 on page 5-44.
Working with Multiple Target Connections

The top-level entries in the left pane list each of the current 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.
- **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.

**Cross-triggering controls**

Expand the processor entries on the Synch tab to see the Trigger controls, shown in Figure 5-29 on page 5-45.
Figure 5-29 Processor Synch tab showing the per-processor controls

The Trigger controls describe communications between the specified processor and other processors in the group:

**In**
Indicates whether the processor responds to the stop requests of other processors.

**Out**
When a processor stops, this indicates whether it makes a stop request to the 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 per-processor check box is unchecked, 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. This means that you can still stop this processor if required, for example, using a breakpoint.

With this system of controlling synchronization you can create master-slave and peer-to-peer synchronization groups. However, you cannot create multiple independent processor groups, where two sets of processors are synchronized within the group but not between the two groups.

**Synchronizing processors**

Figure 5-30 on page 5-46 shows an example Connection Control window with a group of three synchronized processors and their controls.
Working with Multiple Target Connections

In this configuration, each processor can stop all the others, and each processor responds to stop requests. However, none of the processors starts (runs) or single-steps with other processors, because neither the Step nor the Run check boxes are ticked.

**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.

Figure 5-30 Example Connection Control window showing synchronized processors

In this configuration, each processor can stop all the others, and each processor responds to stop requests. However, none of the processors starts (runs) or single-steps with other processors, because neither the Step nor the Run check boxes are ticked.
Appendix A
Setting up the Trace Hardware

This appendix describes how to set up the hardware for the trace configurations supported by RealView Debugger. See Getting started on page 2-8 for details on how trace hardware components interact with RealView Debugger to enable you to perform tracing.

When setting up the trace capture system, do not exceed the timing specifications of the target ETM signals or of the trace capture hardware. See the ARM Multi-Trace User Guide for more information.

This appendix contains the following sections:
- ARM MultiTrace and ARM Multi-ICE on page A-2
- Agilent 16600 or 16700 logic analyzer and Emulation Probe on page A-4
- Agilent 16600 or 16700 logic analyzer and Multi-ICE on page A-8
- Agilent Emulation Probe and Trace Port Analyzer (E5904B) on page A-11
- Tektronix TLA 600 or TLA 700 logic analyzer and Multi-ICE on page A-14.
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 v1.6 Target Configuration Guide.
Figure A-1 Connections for Multi-ICE and Multi-Trace using a separate Multi-ICE server
A.2 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-2 on page A-5:

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-2 on page A-5. 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 needed 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 v1.6 Target Configuration Guide*).
A.3 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-3 on page A-10.

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 needed 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.

  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.

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 v1.6 Target Configuration Guide**).
Figure A-3  Agilent Trace Port Analyzer and Multi-ICE Version 2.2
A.4 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-4 shows the configuration using E5904B components.
Setting up the Trace Hardware

---

**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 Mictor 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-4 on page A-11.

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.
   c. Power-cycle the probe. After about a minute, the prompt `>p` is displayed in the terminal window.
   d. 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](http://www.agilent.com).
d. At the \texttt{\textgreater p} prompt, type \texttt{lan}. 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:
\texttt{lan -i \textit{network-address}}

where \textit{network-address} must be replaced with the dotted-quad network address assigned to the probe. You might also have to set the netmask (using \texttt{lan -s}) and default gateway (using \texttt{lan -g}), depending on the nature of the network. For more details on the \texttt{lan} command, type \texttt{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.

\textbf{Note}

After you have entered this command, you must power-cycle the probe for the change to take effect.

You might also need to change other network parameter settings using the \texttt{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 v1.6 Target Configuration Guide).
A.5  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-5 on page A-15:

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. Plug the A3 and A2 module end into the A3 and A2 connector on the analyzer, and the A1 and A0 module end into the A1 and A0 connector on the analyzer.

   The Tektronix cable is labeled as shown in Table A-1. There are four probe names and colors because Tektronix produce four equivalent but differently labeled connectors.

<table>
<thead>
<tr>
<th>Probe</th>
<th>Colors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tan/orange</td>
<td>A3 and A2 Pin1 (tan) side, A1 and A0 Pin38 (orange) side</td>
</tr>
<tr>
<td>B</td>
<td>Blue/yellow</td>
<td>B3 and B2 Pin1 (tan) side, B1 and B0 Pin38 (orange) side</td>
</tr>
<tr>
<td>C</td>
<td>White/gray</td>
<td>C3 and C2 Pin1 (tan) side, C1 and C0 Pin38 (orange) side</td>
</tr>
<tr>
<td>D</td>
<td>Green/violet</td>
<td>D3 and D2 Pin1 (tan) side, D1 and D0 Pin38 (orange) side</td>
</tr>
</tbody>
</table>

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 v1.6 Target Configuration Guide).

Figure A-5 Tektronix TLA 700 analyzer and Multi-ICE7
Setting up the Trace Hardware
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
- ARM Multi-ICE for XScale on page B-6
- Agilent 16600 or 16700 Logic Analyzer and Emulation Probe on page B-8
- Agilent 16600 or 16700 Logic Analyzer and ARM Multi-ICE on page B-15
- Agilent Trace Port Analyzer and Agilent Emulation Probe on page B-18
- Tektronix TLA 600 or TLA700 and ARM Multi-ICE on page B-21
- ARMulator ETM emulation on page B-24
- Simulators using the Simulator Broker connection on page B-26.

See the RealView Debugger v1.6 Target Configuration Guide for general information on connecting to targets.
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. Configure the Multi-ICE server on the workstation that the Multi-ICE interface unit is connected to.
3. Select Start → Programs → ARM RealView Debugger v1.6.1 → RealView Debugger 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-1.
   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 use Windows Explorer to ensure that files with the extension .dll are not hidden from view.) Click Close to close this dialog.

![Figure B-1 RDI Targets List](image)
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-2).

![Multi-ICE configuration dialog box](image)

**Figure B-2 Multi-ICE configuration dialog box**

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-3 on page B-4).
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 ETM and translate it into the format required by the debugger.

12. 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 use Windows Explorer to ensure that files with the extension .dll are not hidden from view.)

13. 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.
14. Click **OK** to exit the Multi-ICE dialog box.

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.

16. Close the Connection Control window.
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 Debugger v1.6.1 → RealView Debugger 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 use 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.3 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 Debugger v1.6.1, and must be purchased separately.

To configure the Agilent 16600 or 16700 analyzer:

1. Select Start → Programs → ARM RealView Debugger v1.6.1 → RealView Debugger.

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 use 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-5 on page B-9).
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-6 on page B-10).
9. If your target is running in big endian mode, click on **Big-endian**.

10. Click on the **Trace** tab (Figure B-7 on page B-11).
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 use 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-8 on page B-12.
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-9).
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.4 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 Debugger 1.6.1, 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 Debugger v1.6.1 → 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-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 use 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 use 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.5 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 Debugger 1.6.1, and must be purchased separately.

To configure the Agilent Analyzer and probe:

1. Select Start → Programs → ARM RealView Debugger v1.6.1 → RealView Debugger.
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 use 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-5 on page B-9).
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-6 on page B-10.
   b. Click Big-endian.
8. Click Trace to display the trace configuration tab, as shown in Figure B-7 on page B-11.
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 use Windows Explorer to ensure that files with the extension `.dll` are not hidden from view.)

The configured dialog box is shown in Figure B-10.

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**.
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.6 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). 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 Debugger v1.6.1 → RealView Debugger.

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 use 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 Select a new processor.
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-11).

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.

![Figure B-11 Multi-ICE configuration dialog box showing the Trace tab](image)

11. Select the Dragonfly driver `adstla.dll` from the **Select a Trace Capture DLL...** drop-down list:

   - If `adstla.dll` is present in the drop-down list, select it.
   - If `adstla.dll` is not present in the drop-down list, click **Add...** and select `adstla.dll` from the Dragonfly Software TLA installation directory. (You might have to use 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.
12. Click on **Configure**... to display the Dragonfly trace capture configuration dialog box (Figure B-12).

![Figure B-12 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.7 ARMulator ETM emulation

This section describes how to set up and connect to ARMulator.

ARMulator supports emulation of an ARM7TDI or ARM9TDI with an ETM. See the chapter Configuring custom connections in the RealView Debugger v1.6 Target Configuration Guide for more information about configuring ARMulator.

After RealView Debugger and ARMulator are installed, you must configure ARMulator to emulate an ETM. To do this:

1. Select Start → Programs → ARM RealView Debugger v1.6.1 → RealView Debugger to start RealView Debugger.
2. Select File → Connection → Connect to Target... to display the Connection Control window.
4. Right-click on the ARMulator entry and select Configure Device Info... from the context menu to display the ARMulator Configuration dialog box (Figure B-2 on page B-3).
5. In the Processor list, select a processor with a name ending in:
   ETM(L)    A large ETM.
   ETM(M)    A medium-sized ETM.
   ETM(S)    A small ETM.
   Refer to Available resources on page 2-4 for information on the differences between the sizes.

6. Click OK to close the ARMulator Configuration dialog. Close the Connection Control window.

7. Select ARMulator from the targets list in the Connection Control window, by expanding the second-level ARMulator entry, and selecting the processor you are emulating, such as ARM7TDMI-ETM(L).
B.8 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, for example **new_OAK**, to start a simulator connection.
   The connection list expands to show your new connection, for example **SimOAK_1**.
   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 2-40 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.
Glossary

**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.

**ARM Multi-Trace**
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

**ARMulator**
ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

**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.

**ATPCS**
ARM-Thumb Procedure Call Standard.

**Backtracing**
See Stack Traceback.

**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.

**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

CPSR  Current Program Status Register.

  See also Program Status Register.

Debug With Arbitrary Record Format (DWARF)  ARM code generation tools generate debug information in DWARF2 format.

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.

Doubleword  A 64-bit unit of information.

DSP  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.
Execution vehicle

Part of the debug target interface, execution vehicles process requests from the client tools to the target.

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.

FPE

See Floating Point Emulator.

Halfword

A 16-bit unit of information.

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.

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).
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG</td>
<td><em>See Joint Test Action Group.</em></td>
</tr>
<tr>
<td>JTAG interface unit</td>
<td>A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the EmbeddedICE logic and the ETM.</td>
</tr>
<tr>
<td>Little-endian</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><em>See also</em> Big-endian.</td>
</tr>
<tr>
<td>Multi-ICE</td>
<td>A JTAG-based tool for debugging embedded systems.</td>
</tr>
<tr>
<td>Pop-up menu</td>
<td>Also known as <em>Context menu.</em> 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.</td>
</tr>
<tr>
<td>Processor core</td>
<td>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.</td>
</tr>
<tr>
<td>Profiling</td>
<td>Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.</td>
</tr>
<tr>
<td>Program Status Register</td>
<td><em>Program Status Register</em> (PSR), containing some information about the current execution context. It is also referred to as the <em>Current PSR</em> (CPSR), to emphasize the distinction between it and the <em>Saved PSR</em> (SPSR), which records information about an alternate processor mode.</td>
</tr>
<tr>
<td>PSR</td>
<td><em>See</em> Program Status Register.</td>
</tr>
<tr>
<td>RDI</td>
<td><em>See</em> Remote Debug Interface.</td>
</tr>
<tr>
<td>RealView Compilation Tools</td>
<td>RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enable you to write and build applications for the ARM family of RISC processors.</td>
</tr>
<tr>
<td>RealView Debugger Trace</td>
<td>A software product add-on to RealView Debugger that extends the debugging capability with the addition of real-time program and data tracing.</td>
</tr>
<tr>
<td>Remote_A</td>
<td>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.</td>
</tr>
</tbody>
</table>
Remote Debug Interface (RDI)
The Remote Debug Interface 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, ARMulator)
- a debug monitor running on ARM-based hardware accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, Multi-ICE).

RTOS
Real Time Operating System.

RVCT
See RealView Compilation Tools.

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.

**Stack traceback**  
This 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.

**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 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.
Glossary

TPA  See Trace port analyzer.

TLA  See Tektronix Logic Analyzer.

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.

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.

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

Absolute Time  2-76
Access Type   2-76
Access type match, filter on   2-72
Access Type, sort by   2-79
Active connections list   5-16, 5-21
Address as Symbol/Line   2-77
Address as value   2-76
Address column (Analysis window)   2-49
Address expression match, find   2-63
Address expression, match, filter on   2-70
Address mapping   2-42
Address, sort by   2-78
Address/Data   2-26
ADS
  ARMulator   B-24
Agilent
  Emulation module   2-3
  Emulation Probe   A-11, B-18
  ES903A   A-11, A-12

  Logic analyzer   2-3
  Trace Port Analyzer   B-18
  Trace port analyzer   2-4
  16600 or 16700 logic analyzer   A-4, A-8, B-15
  Analysis configuration, selecting   2-41
  Analysis window   2-39
  code window tracking   2-74
  column types   2-48
  Columns menu   2-76
  Context menu   2-83
  Exit   2-81
  Find menu   2-60
  Sort menu   2-78
  Status bar   2-81
  Status lines   2-52
  Tabs   2-47
  update   2-67
  View menu   2-66
  Analyzer properties, configuring   2-12, 2-41
  AND Filters   2-74
  Complex tracepoints

  Trace on X after Y   2-32
  Dialog
    Trace if X after Y   2-33
  Trace on X after Y   2-32
  Angel debug monitor   5-8
  Append new buffer to Existing   2-47
  Architecture, RealView Debugger   5-3, 5-8
  ARM CPU macrocell   2-9
  ARM Developer Suite   ix
  ARM Multi-ICE 2.0 configuration
dialog box   B-3, B-6, B-8, B-15, B-18, B-21, B-24
  ARM processors   2-5
  ETM   2-9
  ARM-ARM-PP connection   4-4
  ARM-A-RR connection   5-16
  ARMulator   2-4, 2-6
  ASIC   2-9, 2-15
  Asterisk, beside connection   5-6
  Asterisk, beside thread   3-9
  ATPCS   3-10
  Attach thread   3-14
Attach Window to a Connection  5-16,  
  5-19
Attaching
  to multiple targets  5-19
Attachment
  of board  5-17
  of window  5-17, 5-21
  status  3-9
Automatic Update on Append to buffer
  2-46
Automatic Update on New buffer  2-46
Auto-range  2-64, 2-70

B
Board, attachment  5-17
Breakpoints/Triggers pane  2-36
Broadcast data links  5-30
Buffer, clear  2-46
Buffer, load from file
  Trace buffer, load from file  2-79
Buffer, Save to file
  Trace buffer, Save to file  2-80
B=>E% column (Analysis window)  
  2-50, 2-77

C
Call-graph  2-58
Capturing profiling information  2-89
CARMETM  A-7, A-9
Changing current connection  5-17
Changing the trace display  2-76
Children of function  2-58
Clear Trace Buffer  2-46
Clearing tracepoints
  2-23, 2-25
Close Loaded File  2-81
Cmd tab, Output pane  5-12
Code view  2-47
Code window
  attachment  3-9, 5-14
  compared to connection  5-4
  create new  5-18
  creating  5-7
  title bar  5-13
Code Window Tracking  2-74
Coherency  5-2
Collect data Around Trigger  2-43, 2-44
Collect data Before Trigger  2-43, 2-44
Color Box, in Code window  5-19
Column types (Analysis window)  2-48
Columns menu (Analysis window)  
  2-76
Commands
  ADDFILE  3-6
  CONNECT  3-7
  RELOAD  3-6
Comments from users xiii
Communication guarding  5-43
Complex tracepoints
  2-2, 2-19, 2-25
  address/data  2-26
  Trace on X after Y executed N times  
  2-33, 2-34
Concepts, multiprocessor debugging  
  5-2
Configure
  analyzer properties  2-41
Configure ETM
  Cycle accurate tracing  2-16
  Enable Timestamps  2-16
  Half-rate clocking enabled  2-14
  Memory map decode  2-15
  Processor stalling  2-15
  Trace buffer size  2-15
  Trace Coprocessor  2-17
  Trace data width  2-13
  Trace port mode  2-13
Configure ETM dialog  2-12
Configure the ETM
  2-10, 2-12
Configure trace capture
  2-10, 2-11, 2-18
Configuring
  Agilent 16600 or 16700 logic
    analyser A-6
  Agilent 16600 or 16700 logic
    analyser B-8
  Analyzer properties
    2-12
  trace options  2-40
Conn tab, Resource Viewer  5-14
Connect Analyzer/Analysis  2-41
Connect tab
  Connection Control window  5-11
  Connecting to a target  2-10, 5-10
  Connecting to DSP  4-2
  Connection
    active list  5-6
asterisk by  5-6
button  5-12
  compared to Code window  5-4
  current  5-6
  drop-down list  5-6
  menu  5-6
  state information  5-4, 5-5, 5-6
Connection button  5-6
Connection Control window  5-11
Connect tab  5-11
  Synch tab  5-11, 5-43, 5-44
Context menu (Analysis window)  2-83
Continue Collecting on Buffer Full  
  2-44
Continuous Collection Mode (no
  Trigger)  2-44
Copy  2-41, 2-84
Count column (Analysis window)  
  2-49, 2-51
Count of hits  2-77
Count, sort by  2-79
Cross-triggering
  5-40, 5-42
  hardware example  5-40
  software example  5-41
  synchronization  5-42
cross-triggering controls  5-44
Cross-triggering example  5-40
Current connection
  5-6, 5-12, 5-17, 5-18
  changing  5-17
  Current thread  3-9
Cycle-accurate tracing
  2-15, 2-16
  Cycle-accurate mode  2-16
  Non cycle-accurate mode  2-17
  Cycles  2-75

D
Data abort example (trace)  2-85
Data synchronization lost  2-53
Data tracing mode  2-14
Data Value in Decimal  2-77
Data Value in Hex  2-77
Data value match, filter on  2-71
Data value match, find  2-64
data value sort by  2-78
Data view  2-47
Data, interpreting  2-47
Index

E

Editing tracepoints 2-36
Elem column (Analysis window) 2-48
Embedded Trace Macrocell. See ETM.
EmbeddedICE logic 2-9
Emulation module firmware versions A-6
Emulation Probe B-18
Emulation Probe, Agilent A-11, B-8, B-18, B-20
Enable Timestamps 2-16
Entry position 2-61, 2-67
Error
  Debug State 2-52
  ETM 2-9, 2-22
  configuration 2-12
  configurations 2-5
  resources 2-4
ETM configuration 2-10
ETM-enabled processors 2-5
Event Triggers, setting and editing 2-42
Example, connecting to three targets 5-19
Executing a program 2-11, 2-38
Execution controls 5-44
Exec% column (Analysis window) 2-50
Exec/B=>E/B=>E Avg column (Analysis window) 2-51, 2-78
Exit Analysis window 2-81
Exit Window 2-81
E5903A, Agilent A-11, A-12

Filter on Address Expression Match 2-70
Filter on Data Value Match 2-71
Filter on Percent Time Match 2-73
Filter on Position Match 2-67
Filter on Raw Address Match 2-69
Filter on Symbol Name Match 2-71
Filter on Time Match 2-68
Filtering captured information 2-66
Filtering trace 2-18
AND filters 2-74
filter on access type match 2-72
filter on address expression match 2-70
filter on data value match 2-71
filter on percent time match 2-73
filter on position match 2-67
filter on raw address match 2-69
filter on symbol name match 2-71
filter on time match 2-68
save filtered buffer to file 2-81
Find Address Expression Match 2-63
Find corresponding source code 2-37
Find Data Value Match 2-64
Find menu 2-60
Find menu (Analysis window) 2-60
Find next 2-84
Find Position Match 2-61
Find previous 2-84
Find Raw Address Match 2-62
Find Symbol Name Match 2-65
Find Time Match 2-61
Find Trigger 2-60
FLEXlm commands 4-3
licenses 1-5
Func view 2-47

F

Feedback xiii
Fetching active threads 3-8
FIFO buffer 2-5, 2-15
FIFO highwater 2-15
FIFO overflow 2-17, 2-53
FIFOFULL logic 2-13
File Editor window 2-18
Files .dll, showing B-16, B-19
Filter on Access Type Match 2-72

G

General Purpose Input/Output. See GPIO.
Getting started with trace 2-8
GPIO 2-13
Index

H
Half-rate clocking 2-14, A-6, A-9, B-13
Hard real-time 3-2
Hardware
  platforms 1-7
  requirements 2-3
  resources 2-4
Hardware solution for trace 2-8
Histogram column (Analysis window) 2-52
Histogram View 2-57
Histogram, logarithmic scale 2-59

I
Images
  target name 3-7
Incorrect synchronization address 2-53
Information about threads 3-13
Information dialog 2-83
Intel StrongARM, and JTAG 5-8
Interpretation of Data 2-77
Interpreting trace data 2-47
Interrupting threads 3-10
IP address B-12, B-18, B-19

J
JTAG
  interface unit 2-3, 2-10
  scan path 5-10
  specification xi

L
last N traces 2-42
Licenses 1-5, 5-2, 5-11
  temporary 1-5
Linear scale 2-59
List selection dialog 2-22
Instat 4-3
Load Trace Buffer from File 2-79
Loading
  an image 2-10, 2-18
trace information 2-79
Locating RTOS data structures 3-6
Logarithmic scale 2-59
Logging commands and output 5-12
Logic Analysis System A-5, A-6, A-9
Logic analyzer B-15
  Agilent 2-4
  automatic configuration B-16
Logic Analyzer Configuration dialog box B-12, B-16
Logic analyzers B-12, B-19
  Agilent 2-3
  Agilent 16600 or 16700 A-4, A-8
  automatic configuration B-13
Multitrace A-2
  Setup Assistant B-13, B-16
  TLA 600 A-14
Loose synchronization 5-40

M
Mastering Regular Expressions xi
MCR 2-17
Memory map decode 2-15
Memory pane 2-18
Micro-Seconds 2-75
Mictor connector A-8, A-12, A-14
Milli-Seconds 2-75
Min/Max B>E column (Analysis window) 2-51, 2-78
Modifying
  local variables 3-10
  shared variables 3-11
  tracepoints 2-37
MRC 2-17
Multi-ICE 2-3, A-2, A-8, B-2, B-6, B-15
Multi-ICE direct connect 4-3, 5-8
Multiple processor license 5-11
Multiple target connections 5-2
Multiple targets, attaching to 5-19
Multiplexed 2-13
Multiprocessor
  architecture 5-3
  communication guarding 5-43
  Connect tab 5-11
  cross-triggering 5-40, 5-42
debugging, concepts 5-2
debugging, facilities 5-2
  loose synchronization 5-40
  second connection 5-5
  Synch tab 5-11, 5-43
  synchronization 5-2, 5-11, 5-38
  synchronization and cross-triggering 5-42
  synchronization facilities 5-43
  synchronization group 5-38
  synchronization skid 5-38
  synchronization skid example 5-39
  synchronization terms 5-38
  synchronizing processors 5-45
  tight synchronization 5-40
Multitrace 2-3, A-2, B-2

N
Name, sort by 2-78
Nano-Seconds 2-75
New Code Window 5-7, 5-19
new_OAK connection 4-3, B-26
Non cycle-accurate mode 2-17
Non-ETM processors 2-6
Non-intrusive 2-2
Normal 2-13

O
Oak
  DSP 4-2
  simulator 1-3, 2-4, 2-7, 4-3
OakDSPCore MaxCore Simulator 1-3, 2-4, 2-7, 4-3
Only connection, disconnecting 5-23
Other column (Analysis window) 2-50
Output pane, Cmd tab 5-12

P
Parents of function 2-58
Parent/Child %ages Relative to Whole
  Time 2-60
Percent time match, filter on 2-73
Physical to Logical Address Mapping 2-42
Pico-Seconds 2-75
Point-to-point data links 5-30
Position 2-76
Position match
  filter on 2-67
  find 2-61
Print Trace Lines 2-81
Procedure Call Standard x
Process Control pane, Thread tab 3-14
Processes
  what are 3-2
  Processor stalling 2-15
    FIFO highwater 2-15
    Stall processor to prevent overflow 2-15
Profile view 2-48
Program execution 2-11, 2-38
Projects
  and window attachment 5-27
  binding in multiprocessor mode 5-27
  bound 5-13
  default binding 5-27
  effects of connecting 5-29
  effects of disconnecting 5-29
  working with multiple connections 5-26
  working with multiple windows 5-27
Prompt dialog 2-63, 2-72

R
Raw address match
  find 2-62
Raw address, match, filter on 2-69
Raw view 2-47
Real-time 2-2
Real-time applications 3-2
RealView Debugger
  architecture 5-8
  documentation suite ix
  extensions 1-2
  single processor license 5-11
  tracing 2-2
  window attachment 5-14
Registers, of thread 3-10
Relative Time 2-76
  sort by 2-79
Remote configuration B-16, B-18
Resource sharing 5-30
Resource Viewer 5-14
  RTOS 3-12
  tabs in 3-12
Resource Viewer Window 3-12, 5-21
  Conn tab 5-14
  Resources 2-4
  Results of disconnecting 5-22
  Reverse Sort 2-78, 2-79
  Ring-buffered last N traces 2-42
RTOS 3-1, 3-2
  Awareness Downloads, goto 1-3, 3-4
  Board/Chip definition files 3-4
  configuring settings 3-4
  data structures 3-12
  debugging 3-3
  loading images 3-7
  processes 3-2
  specifying image target name 3-7
  support library 3-5, 3-7, 3-13
  support package 1-3
  System deadlines 3-3
  Thread button 3-8
  Thread list 3-8, 3-12
  threads 3-3
Set Amount of Trace buffer to read 2-45
Set Trace Buffer Size 2-45
Set Trigger 2-22
Setting disconnect mode 5-24
Setting tracepoints
  complex 2-94
Setup Assistant, in logic analyzer B-13, B-16
Set/Edit Event Triggers 2-42
Set/Toggle Trace Point... 2-21
  Set Trigger 2-22
  Trace End Point 2-23
  Shared memory, as commons link 5-30
  Sharing multiprocessor resources 5-30
  Show code 2-37
  Show Details View 2-74
  Show Position relative to Trigger 2-76
  Simple breakpoints
    setting using Set/Toggle Trace Point 2-20
  Simple trace points
    Set/Toggle Trace Point... 2-21
    Simple tracepoints 2-19
    Simpletrace points 2-2
    Simulator 2-4, 2-6, 2-7
    Single processor RealView Debugger 5-11
    Skid 5-38, 5-42
      example 5-39
    Soft real-time 3-2
    Sort menu (Analysis window) 2-78
    Source tab 2-18
    Stall processor to prevent overflow 2-15
    Start of Excuded Trace Range 2-24
    Start of Trace Range 2-23
    Starting trace capture 2-38
    Status bar (Analysis window) 2-81
    Status lines 2-20, 2-52
    Stop processor on buffer full 2-44
    Stop processor on trigger 2-43
    Stop-Collect-Run profiling (intrusive) 2-42
    Store Control-Flow Changes Only 2-44
    Support for RTOS 1-3
    Supported hardware 1-7
    Suppress data on FIFO full 2-17

Index
Symbol name match
  filter on  2-71
  find  2-65
Symbolic column (Analysis window)  2-49, 2-51
Symmetric Multiprocessor  5-30
Synch tab  5-43, 5-44
  Connection Control window  5-11
Synchronization
  communication guarding  5-43
  cross-triggering  5-40, 5-42
  cross-triggering controls  5-44
  cross-triggering example  5-40
  cross-triggering hardware example  5-40
  cross-triggering software example  5-41
  example  5-45
  Execution controls  5-44
  facilities  5-43
  loose  5-40
  skid  5-38
  skid example  5-39
  terms  5-38
  tight  5-40
Synchronization group  5-38
Synchronization lost  2-53
Synchronization Lost (trace status line)  2-52
Synchronization point  2-52
Synchronized on Run  5-44
Synchronized on Step  5-44
Synchronized on Stop  5-44

T

Tabbed view types  2-47
Target connection  2-10
Target debug interface  5-4, 5-8
Target hardware  5-4
Task Control Block  3-10
TeakLite
  DSP  4-2
  simulator  4-3
Tektronix  B-21
  TLA 600 and 700  A-14
Tektronix logic analyzer  2-4
Temporary license  1-5

Terminology  Glossary-1
The C Programming Language  xi
Thread
  asterisk by  3-9
  attached windows  3-9, 5-17
  attachment  3-14
  changing variables  3-11
  current  3-9
  details of  3-8
  displaying the thread list  3-8
  interrupting  3-10
  registers  3-10
  tab, in Process Control pane  3-14
  threads and processes  3-3
  unattached windows  3-9
Thread button  3-8
Thread list  3-8
  Tight synchronization  5-40
Time match
  filter on  2-68
  find  2-61
  Time Measure from Selected  2-83
  Time Measure from Triggered  2-84
  Timestamps  2-16
  Time/cycl column (Analysis window)  2-48
  Time/Entry, sort by  2-78
  Timing specifications, target  A-1
TLA 600 and 700, Tektronix  A-14
Trace
  address only  2-14
  capture hardware  2-10
  changing the display  2-76
  configuring capture  2-11
  control method  B-11, B-16, B-19
  data and address  2-14
  data only  2-14
  data width  B-13
  examples  2-85
  finding information  2-60
  getting started  2-8
  hardware solution  2-8
  interpreting the data  2-47
  overview  1-2
  using the examples  2-8
  Trace buffer size  2-15
  Trace buffer, clear  2-46
  Trace buffer, save to file  2-81
  Trace capture
    configuring  2-18
    starting  2-38
  Trace capture hardware  2-3
  Trace Coprocessor  2-17
  Trace Data Value Access  2-27
  Trace Data Value Read  2-27
  Trace Data Value Write  2-27
  Trace data width  2-13
  Trace End Point  2-23
  Trace FIFO Overflow (trace status line)  2-52
  Trace hardware requirements  2-3
  Trace if X after Y dialog box  2-32
  Trace if X after ‘n’ executions of Y dialog box  2-33, 2-34
  Trace Instr Exec  2-26
  Trace Instr Fetch  2-26
  Trace on X after Y executed N times  2-33, 2-34
  Trace options
    configuring  2-40
    Trace Pause  2-20
    Trace Pause (trace status line)  2-52
  Trace port analyzer  2-4
  Trace Port Analyzer, Agilent  B-12, B-19
  Trace port mode  2-13
  Tracepoints
    clearing  2-37
    complex  2-2, 2-19, 2-25, 2-94
    disabling  2-37
    editing  2-36
    modifying  2-37
    simple  2-2, 2-19
  Trace, filtering  2-18
  Tracing  2-2
  coprocessors  2-17
  examples  2-85
  procedure  2-10
  saving and loading information  2-79
  Tracing Enabled  2-43
  Tracing fast targets  A-1
  Track in Code Window  2-83
  Trigger controls  5-44
  Trigger, find  2-60
U
Update 2-67
Use Logarithmic Scale for Histogram 2-59
User’s comments xiii

V
Variables, changing in threads 3-11
View menu (Analysis window) 2-66

W
Window attachment 5-17, 5-21
Windows
  attachment 5-14

X
XScale B-6
XScale processor 2-9

Numerics
1st column (Analysis window) 2-50

Symbols
*, beside connection 5-6
*, beside thread 3-9
+Time column (Analysis window) 2-49